2020

Compatibility testing for rooted phylogenetic trees

Yun Deng
Iowa State University

Compatibility testing for rooted phylogenetic trees

by

Yun Deng

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Science

Program of Study Committee:
David Fernandez-Baca, Major Professor
Eulenstein Oliver
Xiaoqiu Huang
Ryan Martin
Wensheng Zhang

The student author, whose presentation of the scholarship herein was approved by the program of study committee, is solely responsible for the content of this dissertation. The Graduate College will ensure this dissertation is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University

Ames, Iowa

2020

TABLE OF CONTENTS

Page

LIST OF FIGURES

ABSTRACT

The tree compatibility problem is a basic special case of the supertree problem. A supertree method is a way to synthesize a collection of phylogenetic trees with partially overlapping taxon sets into a single supertree that represents the information in the input trees. The supertree approach, proposed in the early 90s [5, 6], has been used successfully to build large-scale phylogenies [7].

The original supertree methods were limited to input trees where only the leaves are labeled. We present a new graph-based approach to the following basic problem in phylogenetic tree construction. Let $\mathcal{P} = \{T_1, \ldots, T_k\}$ be a collection of rooted phylogenetic trees over various subsets of a set of species. The tree compatibility problem asks whether there is a phylogenetic tree T with the following property: for each $i \in \{1, \ldots, k\}$, T_i can be obtained from the restriction of T to the species set of T_i by contracting zero or more edges. If such a tree T exists, we say that $\mathcal{P}$ is compatible and that T displays $\mathcal{P}$.

Our approach leads to a $O(M_{\mathcal{P}} \log^2 M_{\mathcal{P}})$ algorithm for the tree compatibility problem, where $M_{\mathcal{P}}$ is the total number of nodes and edges in $\mathcal{P}$. Our algorithm either returns a tree that displays $\mathcal{P}$ or reports that $\mathcal{P}$ is incompatible. Unlike previous algorithms, the running time of our method does not depend on the degrees of the nodes in the input trees. Thus, our algorithm is equally fast on highly resolved and highly unresolved trees.

Semi-labeled trees are phylogenies whose internal nodes may be labeled by higher-order taxa. Thus, a leaf labeled *Mus musculus* could nest within a subtree whose root node is labeled Rodentia, which itself could nest within a subtree whose root is labeled Mammalia. Suppose we are given collection $\mathcal{P}$ of semi-labeled trees over various subsets of a set of taxa. The ancestral compatibility problem asks whether there is a semi-labeled tree $\mathcal{T}$ that respects the clusterings and the ancestor/descendant relationships implied by the trees in $\mathcal{P}$. We give a $\tilde{O}(M_{\mathcal{P}})$ algorithm for the ancestral compatibility problem, where $M_{\mathcal{P}}$ is the total number of nodes and edges in the trees in $\mathcal{P}$. Unlike the best previous algorithm, the running time of our method does not depend on the degrees of the nodes in the input trees.

CHAPTER 1. INTRODUCTION

Building a phylogenetic tree that encompasses all living species is one of the central challenges of computational biology. Two obstacles to achieving this goal are lack of data and conflict among the data that is available. The data shortage is a consequence of the vast disparity in the amount of information at our disposal for different families of species and the limited amount of comparable data across families (20). One approach to overcoming this obstacle begins by identifying subsets of species such that, for each subset, either (a) a reliable phylogeny is already available or (b) there is enough data to build a reliable phylogeny for the subset. The phylogenetic trees for these subsets are then synthesized into a single phylogeny —a *supertree*— for the combined set of species. This approach, proposed in the early 90s (2; 19), has been used successfully to build large-scale phylogenies (see, e.g., (3; 14)).

Any attempt at synthesizing phylogenetic information from multiple input trees must deal with the potential for conflict among these trees. Conflict may arise due to errors, or due to phenomena such as gene duplication and loss, and horizontal gene transfer. A fundamental question is whether conflict exists at all; that is, does there exist a supertree that exhibits the evolutionary relationships implicit in each input tree?

1.1 Leaf-labeled Trees Compatibility Problem

The tree compatibility problem is a basic special case of the *supertree problem*. When only the leaves of input trees are labeled, we can formalize this question as follows. Let $\mathcal{P} = \{T_1, \ldots, T_k\}$ be a collection of rooted phylogenetic trees, where, for each $i \in \{1, \ldots, k\}$, T_i is a phylogenetic tree for a set of species $L(T_i)$. The *tree compatibility problem* asks whether there exists a phylogenetic supertree T for the set of species $\bigcup_{i=1}^{k} L(T_i)$ such that, for each $i \in \{1, \ldots, k\}$, T_i can be obtained from $T|L(T_i)$ — the minimal subtree of T spanning $L(T_i)$ — by zero or more contractions of internal edges. If such a supertree T exists, then we say that T *displays* $\mathcal{P}$ and that $\mathcal{P}$ is *compatible*; otherwise, $\mathcal{P}$ is *incompatible*.

Here we present an algorithm that solves the compatibility problem for rooted trees in $O(M_{\mathcal{P}} \log^2 M_{\mathcal{P}})$ time, where $M_{\mathcal{P}}$ is the total number of vertices and edges in the trees in $\mathcal{P}$. This running time is independent of the degrees of the internal nodes of the input trees.

1.2 Nested Taxa Trees Compatibility Problem

In the *tree compatibility problem* we discussed before, we are given a collection $\mathcal{P} = \{\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_k\}$ of rooted phylogenetic trees with partially overlapping taxon sets. $\mathcal{P}$ is called a *profile* and the trees in $\mathcal{P}$ are the *input trees*. Noe the new question is whether there exists a tree $\mathcal{T}$ whose taxon set is the union of the taxon sets of the input trees, such that $\mathcal{T}$ exhibits the clusterings implied by the input trees. That is, if two taxa are together in a subtree of some input tree, then they must also be together in some subtree of $\mathcal{T}$. The tree compatibility problem has been studied for over three decades (1; 9; 13; 23).

In the original version of the tree compatibility problem, only the leaves of the input trees are labeled. Here we also study a generalization, called *ancestral compatibility*, in which taxa may be *nested*. That is, the internal nodes may also be labeled; these labels represent *higher-order taxa*, which are, in effect, sets of taxa. Thus, for example, an input tree may contain the taxon *Glycine max* (soybean) nested within a subtree whose root is labeled Fabaceae (the legumes), itself nested within an Angiosperm subtree. Note that leaves themselves may be labeled by higher-order taxa. The question now is whether there is a tree $\mathcal{T}$ whose taxon set is the union of the taxon sets of the input trees, such that $\mathcal{T}$ exhibits not only the clusterings among the taxa, but also the ancestor/descendant relationships among taxa in the input trees. Our main result is a $\tilde{O}(M_{\mathcal{P}})$ algorithm for the compatibility problem for trees with nested taxa, where $M_{\mathcal{P}}$ is the total number of nodes and edges in the trees in $\mathcal{P}$.

The rest of the dissertation is organized as follows.

In Chapter 2, we review Leaf-labeled trees and ancestral compatibility problems. In Chapters 3, and 4, we present our approaches for these two type of compatibility problems. In Chapter 5, we conclude this dissertation with a summary of our main contributions and future research plans.

x

CHAPTER 2. LITERATURE REVIEW

A supertree method is a way to synthesize a collection of phylogenetic trees with partially overlapping taxon sets into a single supertree that represents the information in the input trees. The supertree approach, proposed in the early 90s (2; 19), has been used successfully to build large-scale phylogenies (3).

Supertree methods attempt to assemble comprehensive phylogenetic trees — i.e., supertrees — out of smaller trees for restricted sets of taxa (22). There are at least three motivations for the supertree approach. One is speed. Despite major advances in software for inferring phylogenetic trees from sequences (e.g., (33)), analyzing data sets with tens of thousands of species remains challenging. A second motivation is the desire to benefit from the expertise of different researchers on distinct sets of species (14). A third, and perhaps more fundamental, motivation is *partial taxon coverage*. That is, an entire sequence might be missing for a non-negligible subset of the taxa (34).

2.1 Leaf-labeled Compatibility problem

Over three decades ago, Aho et al. (1) laid the foundation for much of the subsequent work on tree compatibility, including ours. Their paper addressed the following question. Suppose we are given a set L of labels and a collection of constraints between pairs of labels, where each constraint has the form $(a, b) \prec (c, d)$. The question is whether there exists a tree T whose leaves are labeled bijectively by L such that if $(a, b) \prec (c, d)$, for some $a, b, c, d \in L$, then the lowest common ancestor of a and b in T is a proper descendant of the lowest common ancestor of c and d in T. Aho et al. devised an algorithm, which they named BUILD, that answers the question in $O(N^2 \log N)$ time, where N is the number of constraints.

The motivation for Aho et al.'s work was not phylogenetics, but relational databases. Steel (23) was perhaps the first to notice the relevance of the BUILD algorithm to tree compatibility. The connection is through *rooted triples*; i.e., rooted phylogenetic trees on three species. Steel observed that triples can be interpreted as lowest common ancestor constraints; thus, the BUILD algorithm can be adapted to determine the com-

patibility of a collection $\mathcal{R}$ of rooted triples in $O(\mathcal{R}|^2)$ time. The BUILD algorithm gives a polynomial-time algorithm for tree compatibility of a collection $\mathcal{P}$ of k phylogenetic trees on n distinct species because (i) $\mathcal{P}$ can be encoded by a collection $\mathcal{R}(\mathcal{P})$ of $O(n^3k)$ rooted triples, obtained by enumerating the restriction of each input tree to every three-element subset of its species set, and (ii) testing the compatibility of $\mathcal{P}$ is equivalent to testing the compatibility of $\mathcal{R}(\mathcal{P})$.

Although it is polynomial, BUILD's running time is unsatisfactory in practice because of the potentially dramatic increase in problem size in going from $\mathcal{P}$ to $\mathcal{R}(\mathcal{P})$. One way to alleviate this is to encode $\mathcal{P}$ with fewer triples. Indeed, $O(n^3k)$ is a naïve estimate on the size of $\mathcal{R}(\mathcal{P})$. The *minimal* set $\mathcal{R}^*$ of rooted triples that encodes $\mathcal{P}$ can be much smaller. If the trees are binary — *fully resolved*, in the language of phylogenetics —, then $O(n)$ triples suffice for each tree, giving us $|\mathcal{R}^*| = O(nk)$. If we allow input trees to have non-binary — that is, *unresolved* — nodes, however, the number of triples needed per input tree is roughly proportional to n^2 (the precise bound depends on the sum of the products of the degrees of internal nodes and the degrees of their children (11)), giving us $|\mathcal{R}^*| = O(n^2k)$. Of course, the extra step of finding $\mathcal{R}^*$ adds to the complexity of the algorithm.

It is also possible to improve the running time of BUILD itself. Henzinger et al. (13) noted that achieving this requires the ability to maintain graph connectivity information dynamically. They devised a data structure that can maintain such information under a series of edge deletions, done in batches, and showed that their data structure leads to an $O(|\mathcal{R}|n^{1/2})$ algorithm to check the compatibility of a collection $\mathcal{R}$ of rooted triples on n distinct species. The running time can be improved to $O(|\mathcal{R}|\log^2 n)$ by using the dynamic graph connectivity data structure of Holm et al. (28).

Aside from the connections with rooted triples, the tree compatibility problem is related to other well-known questions. One of these is the *incomplete directed perfect phylogeny problem* (IDPP), the problem of testing the compatibility of a collection of m "directed partial characters" on n species. We refer the reader to Pe'er et al. (18) for a precise definition of partial characters and IDPP, but note that there is a $\tilde{O}(nm)$ for the problem, which is related to BUILD and relies on dynamic graph connectivity (18). We also note that a collection of k phylogenetic trees on n distinct species corresponds intuitively to a collection of partial characters, where each character encodes the species in the subtree rooted at some node in an input tree. The

correspondence is not, however, exact. Indeed, partial characters from the same tree are related, and must be treated as such.

When the input trees are unrooted, the tree compatibility problem becomes NP-hard (23). Nevertheless, the decision version is polynomial-time solvable if k is fixed (4); that is, the problem is fixed-parameter tractable in k. The proof of fixed-parameter tractability in (4) relies on Courcelle's Theorem (7), and thus is an existence proof, rather than a practical algorithm.

Finally, we note that there are linear-time algorithms for testing the compatibility of a collection of trees that all have exactly the same leaf label set. One such algorithm can be obtained using recent results on computing "loose" and "strict" consensus trees (17). Both types of consensus trees can be found in $O(nk)$ time, which is $O(M_{\mathcal{P}})$ when all leaf label sets are identical.

2.2 Ancestral Compatibility problem

Page (29) was among the first to note the need to handle phylogenies where internal nodes are labeled, and taxa are nested. A major motivation is the desire to incorporate *taxonomies* as input trees in large-scale supertree analyses, as way to circumvent one of the obstacles to building comprehensive phylogenies: the limited taxonomic overlap among different phylogenetic studies (20). Taxonomies group organisms according to a system of taxonomic rank (e.g., family, genus, and species); two examples are the NCBI taxonomy (30) and the Angiosperm taxonomy (32). Taxonomies spanning a broad range of taxa provide structure and completeness that might be hard to obtain otherwise. A recent example of the utility of taxonomies is the Open Tree of Life, a draft phylogeny for over 2.3 million species (14).

Taxonomies are not, strictly speaking, phylogenies. In particular, their internal nodes and some of their leaves are labeled with higher-order taxa. Nevertheless, taxonomies have many of the same mathematical characteristics as phylogenies. Indeed, both phylogenies and taxonomies are *semi-labeled trees* (26; 21). We will use this term throughout the rest of the paper to refer to trees with nested taxa.

The fastest previous algorithm for testing ancestral compatibility, based on earlier work by Daniel and Semple (27), is due to Berry and Semple (25). Their algorithm runs in $O\left(\tau_{\mathcal{P}} \cdot \log^2 n\right)$ time using $O\left(\tau_{\mathcal{P}}\right)$ space. Here, n is the number of distinct taxa in $\mathcal{P}$ and $\tau_{\mathcal{P}} = \sum_{i=1}^{k} \sum_{v \in I(\mathcal{T}_i)} d(v)^2$, where $I(\mathcal{T}_i)$ is the set

of internal nodes of $\mathcal{T}_i$, for each $i \in \{1, \ldots, k\}$, and $d(v)$ is the degree of node v. While the algorithm is polynomial, its dependence on node degrees is problematic: semi-labeled trees can be highly unresolved (i.e., contain nodes of high degree), especially if they are taxonomies.

CHAPTER 3. LEAF-LABELED TREES COMPATIBILITY CHECKING

3.1 Our Contribution

At a high level, our algorithm resembles BUILD (23; 21). There are, however, important differences. BUILD relies on the *triplet graph*, whose nodes are the species and where there is an edge between two species if they are involved in a triplet (see Section 3.2). Our algorithm relies instead on intersection graphs of sets of species associated with certain nodes of the input trees. Our graphs allow a more compact representation of the triplets induced by the trees in $\mathcal{P}$ (see Section 3.4). The key to the correctness of our approach is the close relationship between the triplet graph and our intersection graph. We remark that intersection graphs have a long history of use in testing compatibility, beginning with the work of Buneman (5).

We also take ideas from other sources. From Pe'er et al.'s IDPP algorithm (18), we adapt the idea of a *semi-universal node*. Although the graphs used to solve IDPP and rooted compatibility are different, semi-universal nodes play similar roles in each case: they capture the notion of sets of nodes in the input trees that map to the same node in a supertree, if a supertree exists. The relationship between our algorithm and Pe'er et al.'s goes deeper. Our approach can be viewed as an algorithm for IDPP that takes advantage of the fact that our particular set of incomplete characters arises from a collection of trees.

Intersection graphs are a convenient tool to prove the correctness for our algorithm. They are less convenient for an implementation, because they are hard to maintain dynamically, as our algorithm requires. The difficulty lies in recomputing set intersections whenever the graphs are updated. We avoid this by using *display graphs*, an idea that we borrow from the proof of the fixed-parameter tractability of unrooted compatibility (4). The display graph of a collection $\mathcal{P}$ is obtained by identifying leaves in the input trees that have the same label. Display graphs provide all the connectivity information we need for our intersection graphs (see Lemma 11 of Section 3.5), but are easier to maintain.

Through our techniques, we achieve what, to our knowledge, is the first algorithm for rooted compatibility to achieve near-linear time under all input conditions, regardless of the degrees of the nodes in the input trees. This is an essential quality for dealing with large datasets.

3.2 Preliminaries

For each positive integer r, $[r]$ denotes the set $\{1, \ldots, r\}$.

3.2.1 Phylogenetic Trees

Let T be a rooted tree. We write $V(T)$, $E(T)$, and $r(T)$ to denote the nodes, edges, and the root of T, respectively. For each $x \in V(T)$, we write $\mathrm{Ch}(x)$ and $T(x)$ to denote the set of children of x and the subtree of T rooted at x, respectively. If $\mathrm{Ch}(x) = \emptyset$, we say that x is a *leaf* of T; otherwise, x is an *internal node* of T.

Suppose $u, v \in V(T)$. Then, u is a *descendant* of v if v lies on the path from u to $r(T)$ in T. If u is a descendant of v, then v is an *ancestor* of u. Note that any node in T is a descendant and an ancestor of itself. T is *binary*, or *fully resolved*, if each of its internal nodes has two children.

A (rooted) *phylogenetic tree* is a rooted tree T where every internal node has at least two children, along with a bijection λ that maps each leaf of T to an element of a set of *species*, denoted by $L(T)$. For each $x \in V(T)$, $L(x)$ denotes the set of species mapped to the leaves of $T(x)$; that is, $L(x) = \{\lambda(v) : v \text{ is a leaf in } T(x)\}$. $L(x)$ is called the *cluster* at x. Note that $L(r(T)) = L(T)$. The set of all clusters in T is $\mathrm{Cl}(T) = \{L(x) : x \in V(T)\}$.

The following lemma, adapted from (21, p. 52), is part of the folklore of phylogenetics.

Lemma 1. *Let $\mathcal{H}$ be a collection of non-empty subsets of a set of species X that includes all singleton subsets of X as well as X itself. If there exists a phylogenetic tree T such that $\mathrm{Cl}(T) = \mathcal{H}$, then, up to isomorphism, T is unique.*

Let T be a phylogenetic tree and A be a set of species. The *restriction* of T to A, denoted $T|A$ is the phylogenetic tree with species set $L(T) \cap A$ where $\mathrm{Cl}(T|A) = \{C \cap A : C \in \mathrm{Cl}(T) \text{ and } C \cap A \neq \emptyset\}$. Let

T' be a phylogenetic tree. T *displays* T' if $\mathrm{Cl}(T') \subseteq \mathrm{Cl}(T|L(T'))$. Equivalently, T displays T' if T' can be obtained from $T|L(T')$ by zero or more contractions of internal edges.

A *rooted triple* is a binary phylogenetic tree on three leaves. A rooted triple with leaves a, b, and c is denoted $ab|c$ if the path from a to b does not intersect the path from c to the root. We treat $ab|c$ and $ba|c$ as equivalent.

When restricted to the three-element subsets of its species set, a phylogenetic tree T induces a set $\mathcal{R}(T)$ of rooted triples, defined as $\mathcal{R}(T) = \{T|X : X \subseteq L(T), |X| = 3 \text{ and } T|X \text{ is binary}\}$.

Lemma 2 ((21, p. 119))**.** *Let T and T' be two phylogenetic trees. Then T displays T' if and only if $\mathcal{R}(T') \subseteq \mathcal{R}(T)$.*

3.2.2 Profiles and Compatibility

Throughout the rest of this paper $\mathcal{P} = \{T_1, \ldots, T_k\}$ denotes a set where, for each $i \in [k]$, T_i is a phylogenetic tree. We refer to $\mathcal{P}$ as a *profile*, and write $L(\mathcal{P})$ to denote $\bigcup_{i\in[k]} L(T_i)$, the *species set* of $\mathcal{P}$. We write $V(\mathcal{P})$ for $\bigcup_{i\in[k]} V(T_i)$, $E(\mathcal{P})$ for $\bigcup_{i\in[k]} E(T_i)$, and $\mathcal{R}(\mathcal{P})$ for $\bigcup_{i\in[k]} \mathcal{R}(T_i)$. Given a subset A of $L(P)$, $\mathcal{P}|A$ denotes the profile $\{T_1|A, \ldots, T_k|A\}$. The *size* of $\mathcal{P}$ is $M_{\mathcal{P}} = |V(\mathcal{P})| + |E(\mathcal{P})|$. Note that $M_{\mathcal{P}} = O(nk)$, where, as before, $n = |L(P)|$, the number of species.

Profile $\mathcal{P}$ is *compatible* if there exists a phylogenetic tree T such that $L(T) = L(\mathcal{P})$ and, for each $i \in [k]$, T displays T_i. If such a tree T exists, we say that T *displays* $\mathcal{P}$. See Figures 3.1 and 3.2.

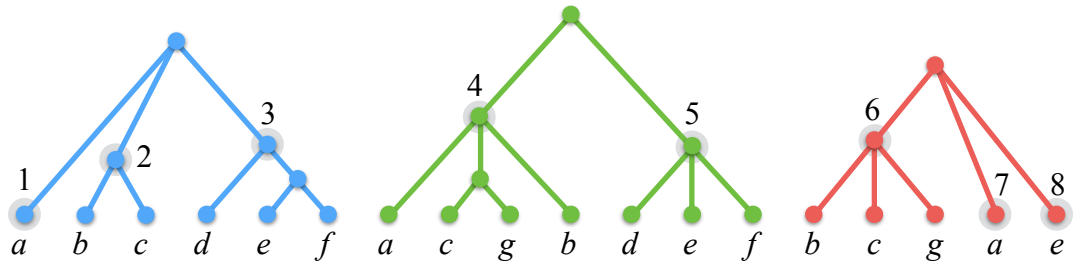

Figure 3.1 A profile $\mathcal{P} = \{T_1, T_2, T_3\}$, where $L(\mathcal{P}) = \{a, b, c, d, e, f, g\}$. Certain nodes have been numbered for reference in Figures 3.5 and 3.7.

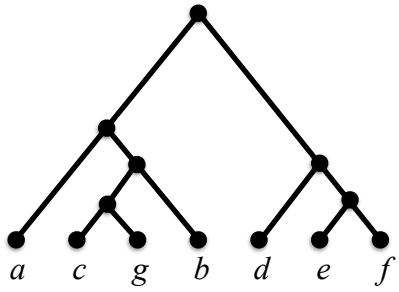

Figure 3.2 A tree that displays the profile $\mathcal{P}$ of Figure 3.1.

3.3 Three Graphs

Here we introduce three important graphs. The first is the triple graph, which is the basis for Semple and Steel's version of the BUILD algorithm. The second is the cluster intersection graph, which describes the intersection patterns among clusters associated with a collection of nodes in $\mathcal{P}$. We show that the cluster intersection graph, for certain sets of clusters, offers the same information as the triple graph. The cluster intersection graph is the basis for BUILDST, our version of the BUILD algorithm, to be described in Section 3.4. Finally, we define the display graph, which offers the same information as the cluster intersection graph, but is easier to maintain dynamically.

We use standard graph terminology. In particular, the *connected components* of a graph are the equivalence classes of vertices under the "is reachable from" relation (6, p. 1170).

3.3.1 The Triple Graph

The *triple graph* of a profile $\mathcal{P}$, denoted $\Gamma(\mathcal{P})$, is the graph whose vertex set is $L(P)$ and where there is an edge between species a and b if and only if there exists a $c \in L(P)$ such that $ab|c \in \mathcal{R}(\mathcal{P})$. See Figure 3.3.

The following fact concerning singleton profiles will be useful.

Lemma 3. *Let T be a phylogenetic tree, and let $u_1, \ldots, u_p$ be the children of $r(T)$. Then, the connected components of $\Gamma(\{T\})$ are $L(u_1), \ldots, L(u_p)$, where $p \geq 2$.*

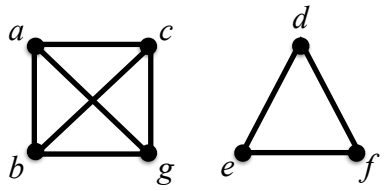

Figure 3.3 The triple graph $\Gamma(\mathcal{P})$ for the profile $\mathcal{P}$ of Figure 3.1. Its connected components are $A_1 = \{a, b, c, g\}$ and $A_2 = \{d, e, f\}$.

Proof. First, note that, by definition, $\Gamma(\{T\})$ cannot have an edge between any species $a \in L(u_j), b \in L(u_h)$, where $j, h \in [p]$, $j \neq h$.

Now, consider any $j \in [p]$. We claim that $L(u_j)$ is a clique in $\Gamma(\{T\})$. This is trivially true if $|L(u_j)| = 1$. Suppose, therefore, that $|L(u_j)| \geq 2$. Consider any two distinct species $a, b \in L(u_j)$, and suppose $c \in L(u_h)$, for some $h \in [p]$, $h \neq j$. Then, we must have $ab|c \in \mathcal{R}(\{T\})$, so there is an edge between a and b in $\Gamma(\{T\})$. We conclude that $L(u_j)$ is a clique in $\Gamma(\{T\})$. □

Given a subset A of $L(\mathcal{P})$, $\Gamma(\mathcal{P}|A)$ denotes the subgraph of $\Gamma(\mathcal{P})$ induced by the leaf set A. $\Gamma(\mathcal{P}|A)$ could have a single connected component or several connected components.

The triple graph plays a fundamental role in the BUILD algorithm (21, p. 119). For completeness, we give the pseudocode for BUILD in Algorithm 1, slightly adapted from Semple and Steel's description. The two trivial cases of BUILD, occur when the input profile has one or two species; these cases are handled by Lines 2–10. Lines 11–20 are the core of BUILD. It can be shown that, if the triple graph of $\mathcal{P}$ is connected, $\mathcal{P}$ must be incompatible (indeed, the proof of this fact is implicit in the proof of Lemma 10, Section 3.3.2.2). Otherwise, the triple graph breaks down into two or more components, $A_1, \ldots, A_p$, and it can be shown that $\mathcal{P}$ is compatible if and only if $\mathcal{P}|A_j$ is compatible for each $j \in [p]$. Lines 14-19 consider each of these subproblems recursively. If all the recursive calls succeed in constructing trees, Line 20 assembles these into a tree that displays $\mathcal{P}$.

3.3.2 The Cluster Intersection Graph

Instead of the triple graph, our version of BUILD uses the *cluster intersection graph*, a graph that reflects the intersection patterns among the clusters at certain sets of nodes in $V(\mathcal{P})$, called *positions*. It is well-

Algorithm 1: BUILD($\mathcal{P}$)

Input: A profile $\mathcal{P}$.

Output: A tree T that displays $\mathcal{P}$, if $\mathcal{P}$ is compatible; `incompatible` otherwise.

Create a node $r_{\mathcal{P}}$;
if $|L(\mathcal{P})| = 1$ **then**
    Let ℓ be the label in $L(\mathcal{P})$;
    return the tree consisting of node $r_{\mathcal{P}}$, labeled by ℓ;
if $|L(\mathcal{P})| = 2$ **then**
    Let ℓ_1, ℓ_2 be the two labels in $L(\mathcal{P})$;
    foreach $j \in [2]$ **do**
        Create a node r_j, labeled ℓ_j ;
        Make $r_{\mathcal{P}}$ the parent of r_j;
    return the tree with root $r_{\mathcal{P}}$;
Let $A_1, A_2, \ldots, A_p$ be the connected components of $\Gamma(\mathcal{P})$
if $p = 1$ **then**
    return `incompatible`;
foreach $j \in [p]$ **do**
    Let $t_j = \text{BUILD}(\mathcal{P}|A_j)$;
    if t_j *is a tree* **then**
        Make $r_{\mathcal{P}}$ the parent of $r(t_j)$
    else
        return `incompatible`
return the tree with root $r_{\mathcal{P}}$;

known that clusters and triples provide equivalent information (21) — in fact, in Subsection 3.2.1, we saw that one can define the notion of "displays" via clusters or triples. On the other hand, clusters can sometimes provide the needed information more compactly than triples, since a single cluster can correspond to multiple triples; i.e., every pair of species in a cluster forms a triple with every species outside the cluster.

Next, we define positions, valid positions, and cluster intersection graphs formally. We then introduce two concepts that are essential to our compatibility algorithm: the notion of a *semi-universal node* in a position and the notion of the *successor* of a position.

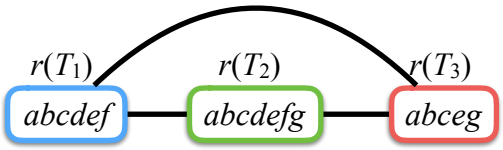

Figure 3.4 The cluster intersection graph $G_{\mathcal{P}}(U_{\text{init}})$ for profile $\mathcal{P}$ of Figure 3.1. In this figure and the next, labels inside each node are the elements of the cluster at that node.

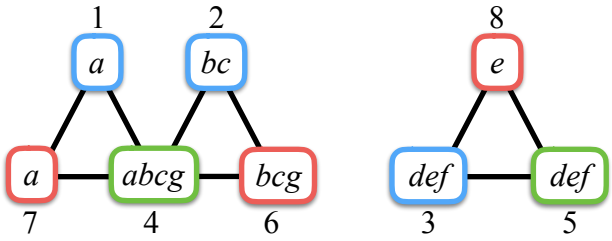

Figure 3.5 The cluster intersection graph $G_{\mathcal{P}}(U)$ for valid position $U = \{1, 2, 3, 4, 5, 6, 7, 8\}$ in profile $\mathcal{P}$ of Figure 3.1. The connected components of $G_{\mathcal{P}}(U)$ are $W_1 = \{1, 2, 4, 6, 7\}$ and $W_2 = \{3, 5, 8\}$. Note that $L(W_1) = \{a, b, c, g\} = A_1$ and $L(W_2) = \{d, e, f\} = A_2$, where A_1 and A_2 are the connected components of the triple graph of Figure 3.3.

3.3.2.1 Positions and Valid Positions

Let U be a subset of $V(\mathcal{P})$; we refer to any such subset as a *position* in $\mathcal{P}$. A position of special interest is the *initial position*, denoted U_{init}, defined as follows

$$U_{\text{init}} = \{r(T_i) : i \in [k]\}. \tag{3.1}$$

The *cluster intersection graph* for position U, denoted $G_{\mathcal{P}}(U)$, is the intersection graph of the clusters associated with the nodes in U. That is, $G_{\mathcal{P}}(U)$ is the graph whose vertex set is U and where $u, v \in U$ are joined by an edge if and only if $L(u) \cap L(v) \neq \emptyset$. See Figures 3.4 and 3.5.

The *species set* of a position U in $\mathcal{P}$, denoted $L(U)$, is defined as $L(U) = \bigcup_{u \in U} L(u)$. For each $i \in [k]$, let $U(i) = U \cap V(T_i)$.

A position U is *valid* if, for each $i \in [k]$,

(V1) if $|U(i)| \geq 2$, then there exists a node $v \in V(T_i)$ such that $U(i) \subseteq \text{Ch}(v)$ and

(V2) $L(U(i)) = L(T_i) \cap L(U)$.

The initial position U_{init} trivially satisfies property (V1). For each $i \in [k]$, we have $L(U_{\text{init}}(i)) = L(T_i)$, and, since $L(U_{\text{init}}) = L(\mathcal{P})$, we have $L(T_i) \cap L(U_{\text{init}}) = L(T_i)$. Hence, $L(U_{\text{init}})$ satisfies property (V2), and, therefore, U_{init} is valid.

Consider the position U illustrated in Figure 3.5. For each $i \in [k]$, $U(i)$ consists of the children of $r(T_i)$; hence, U satisfies (V1). Since $L(U) = L(\mathcal{P})$ and, for each $i \in [k]$, $L(U(i)) = L(T_i)$, property (V2) holds as well. Therefore, U is valid.

Figure 3.5 illustrates that $G_{\mathcal{P}}(U)$ may have more than one connected component. Observe that each connected component W_1 and W_2 in that figure is itself valid. To verify that W_1 is valid, note first that $L(W_1) = \{a, b, c, g\}$, and that $W_1(1) = \{1, 2\}, W_1(2) = \{4\}, W_1(3) = \{6, 7\}$. Position W_1 clearly satisfies property (V1); it also satisfies property (V2), since

$$
\begin{aligned}
L(W_1(1)) &= \{a, b, c\} = \{a, b, c, d, e, f\} \cap \{a, b, c, g\} = L(T_1) \cap L(W_1), \\
L(W_1(2)) &= \{a, b, c, g\} = \{a, b, c, d, e, f, g\} \cap \{a, b, c, g\} = L(T_2) \cap L(W_1), \qquad \text{and} \\
L(W_1(3)) &= \{a, b, c, g\} = \{a, b, c, e, g\} \cap \{a, b, c, g\} = L(T_3) \cap L(W_1).
\end{aligned}
$$

One can similarly verify that W_2 is valid. The preceding example illustrates a general fact, expressed formally in the next lemma.

Lemma 4. *Let $W_1, \ldots, W_p$ be the connected components of $G_{\mathcal{P}}(U)$, for some valid position U. For each $j \in [p]$, W_j is a valid position.*

Proof. Since U satisfies (V1), so does W_j, for each $j \in [p]$. Since U satisfies (V2), $L(U(i)) = L(T_i) \cap L(U)$, for each $i \in [k]$. Now, for any $h \in [p]$, $h \neq j$, we have $L(W_j) \cap L(W_h) = \emptyset$, and thus $L(W_j(i)) \cap L(W_h(i)) = \emptyset$ for any $i \in [k]$. Thus, $L(W_j(i)) = L(T_i) \cap L(W_j)$, so W_j satisfies (V2). Therefore, W_j is a valid position. □

Together with Lemma 1, the next result shows that, for any valid position U and any $i \in [k]$, $T_i|L(U)$ is essentially determined by the descendants of $U(i)$.

Lemma 5. *If U is a valid position in $\mathcal{P}$, then, for each $i \in [k]$, $\text{Cl}(T_i|L(U)) = \{L(U(i))\} \cup \{L(v) : v$ is a descendant of a node in $U(i)\}$.*

Proof. By properties (V1) and (V2), for each $i \in [k]$, every cluster of $T_i|L(U)$ — except, possibly, the cluster at the root of $T_i|L(U)$ — is a cluster at a descendant of some node in $U(i)$. The lemma follows. □

A valid position U of $V(\mathcal{P})$ is *compatible* if there exists a phylogenetic tree T with $L(T) = L(U)$ that displays $T_i|L(U)$ for every $i \in [k]$. If such a tree T exists, we say that T *displays* U.

Lemma 6. *Profile $\mathcal{P}$ is compatible if and only if every valid position U in $\mathcal{P}$ is compatible.*

Proof. (Only if) Suppose $\mathcal{P}$ is compatible, but there is a valid position U of $\mathcal{P}$ that is not compatible. Let T be a tree that displays $\mathcal{P}$. But then $T|L(U)$ displays U, a contradiction.

(If) Suppose every valid position in $\mathcal{P}$ is compatible. Then, in particular, so is the initial position, U_{init}. Let T be a tree that displays U_{init}. Thus, by definition, T displays $T_i|L(U_{\text{init}})$, for every $i \in [k]$. Since $L(T) = L(U_{\text{init}}) = L(\mathcal{P})$, we have that $T_i|L(U_{\text{init}}) = T_i$, and thus T displays T_i, for every $i \in [k]$. Hence, $\mathcal{P}$ is compatible. □

3.3.2.2 Semi-Universal Nodes and Successor Positions

Let U be a valid position in $\mathcal{P}$. A node $v \in U$ is *semi-universal* if there is an $i \in [k]$ such that v is an internal node in T_i and $U(i) = \{v\}$. We write $S(U)$ to denote the set of all semi-universal nodes in U.

Note that every node in U_{init} is semi-universal; i.e., $S(U_{\text{init}}) = U_{\text{init}}$. Consider again the graph $G_{\mathcal{P}}(U)$ of Figure 3.5, whose connected components are W_1 and W_2. As we saw earlier, $W_1(1) = \{1, 2\}, W_1(2) = \{4\}, W_1(3) = \{7, 6\}$. Thus, $S(W_1) = \{4\}$. On the other hand, $W_2(1) = \{3\}, W_2(2) = \{5\}, W_2(3) = \{8\}$. Thus, $S(W_2) = \{3, 5\}$. Node 8 is not semi-universal in W_2, since 8 is a leaf in T_3.

The term "semi-universal", borrowed from Pe'er et al. (18), derives from the following fact. Suppose that $\mathcal{P}$ is compatible, that T is a supertree that displays $\mathcal{P}$, and that U is a valid position in $\mathcal{P}$. Then, intuitively, the nodes of $S(U)$ map to a node w in T such that $L(w) = L(U)$. (The precise sense in which this is true is stated formally in the proof of Theorem 1 of Section 3.4.) Thus, the nodes in $S(U)$ are "almost" universal ancestors for all the species in $L(U)$.

The *successor* of a valid position U is the result of replacing each semi-universal node in U by its children. That is, the successor of U is $(U \setminus S(U)) \cup \bigcup_{v \in S(U)} \text{Ch}(v)$. If $S(U)$ is empty, then the successor of U is U itself. In Figure 3.1, $U = \{1, 2, 3, 4, 5, 6, 7, 8\}$ is the successor of U_{init}.

Lemma 7. *Let U be the successor of some valid position in a profile $\mathcal{P}$. Then, U is a valid position that contains no semi-universal nodes.*

Proof. Suppose U is the successor of valid position U'. Each element of U is either an element of U' or a child of some semi-universal node $v \in S(U')$. In the latter case, every child of v is in U'. Since U' is valid, and for every non-leaf node v, $L(v) = \bigcup_{w \in \mathrm{Ch}(v)} L(w)$, U must also be valid. Since $|\mathrm{Ch}(v)| \neq 1$, for every $v \in V(\mathcal{P})$, we have $|U(i)| \neq 1$ for each $i \in [k]$. Thus, U contains no semi-universal nodes. □

As illustrated by Figures 3.4 and 3.5, even if the cluster intersection graph for a valid position U is connected, this graph may be disconnected for the successor of U. Each connected component of the latter graph has its own, possibly empty, set of semi-universal nodes.

The main result of this section, Lemma 10, is that if U is a valid position with no semi-universal nodes and $G_{\mathcal{P}}(U)$ is connected, then it must be the case that $\mathcal{P}$ is incompatible. As we shall see in Section 3.4, this fact is crucial to the correctness of our compatibility algorithm. The result is a consequence of the close relationship between the triple graph and the cluster intersection graph, explored in the next two lemmas.

Lemma 8. *Suppose that U is a valid position in $\mathcal{P}$ that contains no semi-universal nodes. Let a and b be any two species in $L(U)$. Then, (a, b) is an edge in $\Gamma(\mathcal{P}|L(U))$ if and only if there exists a node $v \in U$ such that $a, b \in L(v)$.*

Proof. First, note that, since U has no semi-universal nodes, $|U(i)| \neq 1$, for each $i \in [k]$.

(Only if) Suppose that (a, b) is an edge in $\Gamma(\mathcal{P}|L(U))$. Then, there is an $i \in [k]$ such that $ab|x \in \mathcal{R}(T_i|L(U))$. Thus, there must be a proper descendant w of $r(T_i|L(U))$ such that $\{a, b\} \subseteq L(w)$. Lemma 5 and the fact that $|U(i)| > 1$ imply that $L(w) \subseteq L(v)$ for some $v \in U(i)$.

(If) Suppose that there is an $i \in [k]$ such that $a, b \in L(v)$ for some $v \in U(i)$. Choose a node $v' \in U(i) \setminus \{v\}$ — such a v' must exist, since $|U(i)| \geq 2$ — and choose some $x \in L(v')$. Then, $ab|x \in \mathcal{R}(T_i|L(U))$, and, hence, (a, b) is an edge of $\Gamma(\mathcal{P}|L(U))$. □

Lemma 9. *Suppose that U is a valid position in $\mathcal{P}$ that contains no semi-universal nodes. Let $W_1, \ldots, W_p$ be the connected components of $G_{\mathcal{P}}(U)$. Then, the connected components of $\Gamma(\mathcal{P}|L(U))$ are precisely $L(W_1), \ldots, L(W_p)$.*

Proof. Let Π_1 and Π_2 be defined as follows.

$$\Pi_1 = \{A : A \text{ is a connected component of } \Gamma(\mathcal{P}|L(U))\}$$

$$\Pi_2 = \{L(W) : W \text{ is a connected component of } G_{\mathcal{P}}(U)\}.$$

Both Π_1 and Π_2 are partitions of $L(U)$. We prove that $\Pi_1 = \Pi_2$ by showing that (a) for each connected component A of $\Gamma(\mathcal{P}|L(U))$ there exists a connected component W of $G_{\mathcal{P}}(U)$ such that $A \subseteq L(W)$, and (b) for each connected component W of $G_{\mathcal{P}}(U)$ there exists a connected component A of $\Gamma(\mathcal{P}|L(U))$ such that $L(W) \subseteq A$.

(a) Let A be any connected component of $\Gamma(\mathcal{P}|L(U))$. We argue that any two species a, b in A must be in the same connected component of $G_{\mathcal{P}}(U)$. Let $U_a = \{v \in U : a \in L(v)\}$ and $U_b = \{v \in U : b \in L(v)\}$. Then, each of U_a and U_b is a clique in $G_{\mathcal{P}}(U)$. It thus suffices to show that there is a path between some node in U_a and some node in U_b.

By the definition of A, there exists a path between a and b in $\Gamma(\mathcal{P}|L(U))$. Suppose this path is $\rho = \langle a_1, \ldots, a_m \rangle$, where $a_1 = a$ and $a_m = b$. By Lemma 8, for each $l \in [m-1]$, there exists a node $w_l \in U$ such that $\{a_l, a_{l+1}\} \subseteq L(w_l)$. For each $l \in [m-2]$, $L(w_l) \cap L(w_{l+1}) \neq \emptyset$, so, either $w_l = w_{l+1}$ or there is a edge between w_l and w_{l+1} in $G_{\mathcal{P}}(U)$. Let $\pi = \langle w_1, \ldots, w_{m-1} \rangle$. Then, we can extract from π a subsequence that is a path from w_1 to w_{m-1} in $G_{\mathcal{P}}(U)$. By the definition of ρ, $a \in L(w_1)$ and $b \in L(w_{m-1})$, so $w_1 \in U_a$ and $w_l \in U_b$. This completes the proof of part (a).

(b) Let W be any connected component of $G_{\mathcal{P}}(U)$. If $|L(W)| = 1$, the statement holds trivially, so assume that $|L(W)| > 1$. We argue that any two species a, b in $L(W)$ are in the same connected component of $\Gamma(\mathcal{P}|L(U))$. Let v_a and v_b be nodes in W such that $a \in L(v_a)$ and $b \in L(v_b)$. If $v_a = v_b$, then, by Lemma 8, (a, b) is an edge of $\Gamma(\mathcal{P}|L(U))$, and we are done. So, suppose instead that $v_a \neq v_b$.

Let us call a path π from v_a to v_b *good* if $|L(w)| > 1$ for every node w in π. We claim that there exists a good path from v_a to v_b. To prove this claim, we first argue that we can choose v_a and v_b such that $|L(v_a)|, |L(v_b)| > 1$. Indeed, consider the case of species a (the case for b is analogous). If $|L(v)| = 1$ for every node $v \in W$ such that $a \in L(v)$, then we would have $|L(W)| = 1$, contradicting our assumption that $|L(W)| > 1$. Now, suppose the path π from v_a to v_b has a node $w \notin \{v_a, v_b\}$ such that $|L(w)| = 1$. Let w' and w'' be the predecessor and successor of w in π. Then, $L(w') \cap L(w'') = L(w) \neq \emptyset$, so there is an edge

between w' and w''. Thus, we can delete w from π and the resulting sequence remains a path between v_a and v_b.

Let $\pi = \langle w_1, \ldots, w_l \rangle$, where $w_1 = v_a$ and $w_l = v_b$, be a good path from v_a to v_b in $G_{\mathcal{P}}(U)$. Choose a sequence of species $\rho = \langle c_1, \ldots, c_{l+1} \rangle$, where $c_1 = a$, $c_{l+1} = b$ and, for each $j \in [l]$, $c_j, c_{j+1} \in L(w_j)$ and $c_j \neq c_{j+1}$. Note that such a choice is always possible. Then, by Lemma 8, (c_j, c_{j+1}) is an edge of $\Gamma(\mathcal{P}|L(U))$. Hence, ρ is a path from a to b in $\Gamma(\mathcal{P}|L(U))$. □

Lemma 10. *Let U be a valid position in $\mathcal{P}$ such that U contains no semi-universal nodes. Then, if $G_{\mathcal{P}}(U)$ is connected, U is incompatible.*

Proof. Suppose, for contradiction, that U is compatible. Then, there exists a phylogenetic tree T_U that displays U. By Lemma 3, $\Gamma(\{T_U\})$ has at least two connected components A and B. By Lemma 9, however, $\Gamma(\mathcal{P}|L(U))$ is connected, so there exist species $a \in A$ and $b \in B$ such that $ab|c \in \mathcal{R}(\mathcal{P}|U)$. But $ab|c \notin \mathcal{R}(T_U)$, and, by Lemma 2, T_U does not display some tree in $\mathcal{P}|L(U)$, a contradiction. □

3.3.3 The Display Graph

The *display graph* of $\mathcal{P}$, denoted H_P, is the graph constructed as follows. For each species $\ell \in L(\mathcal{P})$, create a new node $x_\ell \notin V(\mathcal{P})$, and let

$$X_{\mathcal{P}} = \{x_\ell : \ell \in L(\mathcal{P})\}. \tag{3.2}$$

Then, $H_{\mathcal{P}}$ is the graph whose vertex set is $V(\mathcal{P}) \cup X_{\mathcal{P}}$ and whose edge set is $E(\mathcal{P}) \cup \{(u, x_\ell) : u$ is a leaf in T_i, for some $i \in [k]$, such that $\lambda(u) = \ell\}$. See Figure 3.6. Note that $H_{\mathcal{P}}$ has $O(M_{\mathcal{P}})$ nodes and edges, and can be constructed from $\mathcal{P}$ in $O(M_{\mathcal{P}})$ time.

We remark that the display graph is usually defined as the result of identifying leaves in $\mathcal{P}$ labeled by the same species (4). Contrast this with $H_{\mathcal{P}}$, which connects leaves with a common label through nodes in $X_{\mathcal{P}}$. Even though our definition of the display graph is slightly non-standard, the difference with the standard one is minor, and only serves to simplify our presentation.

Given a valid position U in $\mathcal{P}$, we define $H_{\mathcal{P}}(U)$ as the subgraph of $H_{\mathcal{P}}$ induced by the set $\{v : v$ is a descendant of some node $u \in U\} \cup \{x_\ell \in X_{\mathcal{P}} : \ell \in L(U)\}$. Note that $H_{\mathcal{P}}(U_{\text{init}}) = H_{\mathcal{P}}$. The next result

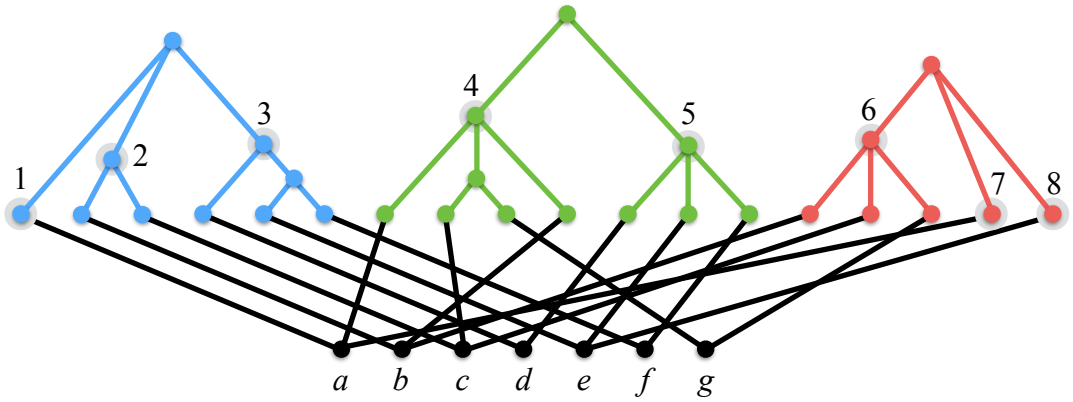

Figure 3.6 The display graph $H_{\mathcal{P}}$ for the profile $\mathcal{P}$ of Figure 3.1. Nodes in the set $\{x_s : s \in L(\mathcal{P})\}$ are labeled with the corresponding species. Species labeling the leaves of trees in $\mathcal{P}$ are omitted.

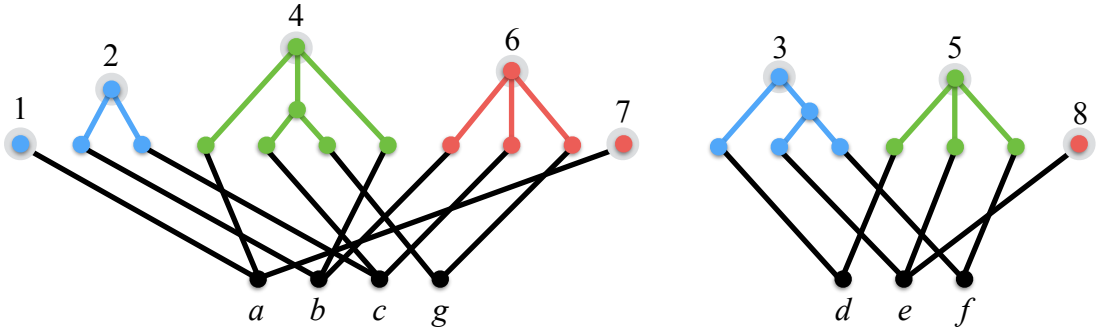

Figure 3.7 The graph $H_{\mathcal{P}}(U)$ for the profile $\mathcal{P}$ of Figure 3.1 and the valid position $U = \{1, 2, 3, 4, 5, 6, 7, 8\}$. The leaf label sets in each of the two components are in one-to-one correspondence with those in $\Gamma(\mathcal{P})$ (Figure 3.3) and $G_P(U)$ (Figure 3.5).

states the basic properties of $H_{\mathcal{P}}(U)$; in particular, part (ii) indicates the close relationship between $H_{\mathcal{P}}(U)$ and $G_{\mathcal{P}}(U)$. Figure 3.7 illustrates this result.

Lemma 11. *The following statements hold for any valid position U of $V(\mathcal{P})$.*

(i) Let v be a node in U such that v is an internal node in some tree in $\mathcal{P}$. If $U' = (U \setminus \{v\}) \cup \mathrm{Ch}(v)$, then $H_{\mathcal{P}}(U')$ is obtained from $H_{\mathcal{P}}(U)$ by deleting v and every edge (v, u) such that $u \in \mathrm{Ch}(v)$.

(ii) Let v and w be any two nodes in U. Then, v and w are in the same connected component in $G_{\mathcal{P}}(U)$ if and only if they are in the same connected component of $H_{\mathcal{P}}(U)$.

Proof. The proof of part (i) is trivial, so we focus on part (ii).

(Only if) Suppose v and w are in the same connected component of $G_{\mathcal{P}}(U)$. Then, there exists a path between v and w in $G_{\mathcal{P}}(U)$. Let $\langle u_1, \ldots, u_m \rangle$ be one such path, where $u_1 = v$ and $u_m = w$. By definition of $G_{\mathcal{P}}(U)$, for each $i \in [m-1]$, $L(u_i) \cap L(u_{i+1}) \neq \emptyset$. Pick any species $s \in L(u_i) \cap L(u_{i+1})$. Hence, in $H_{\mathcal{P}}(U)$ there is a path from u_i to u_{i+1} that goes through x_s. Since this holds for each $i \in [m-1]$, there exists a path from $u_1 = v$ to $u_m = w$ in $H_{\mathcal{P}}(U)$. Thus, v and w are in the same connected component in $H_{\mathcal{P}}(U)$.

(If) Suppose v and w are in the same connected component of $H_{\mathcal{P}}(U)$. Then, there exists a path π between v and w in $H_{\mathcal{P}}(U)$. Let $\pi' = \langle x_{\ell_1}, \ldots, x_{\ell_m} \rangle$, where $\{\ell_1, \ldots, \ell_m\} \subseteq L(\mathcal{P})$, be the subsequence of π obtained by striking out from π all nodes not in $X_{\mathcal{P}}$. Note that $x_{\ell_1} \in L(v)$ and $x_{\ell_m} \in L(w)$. Hence, if $m = 1$, we have $\ell_1 \in L(v) \cap L(w)$, so there is an edge between v and w in $G_{\mathcal{P}}(U)$, and, consequently, v and w are in the same connected component of $G_{\mathcal{P}}(U)$. Thus, suppose $m > 1$. Let $u_0 = v$ and $u_m = w$. By construction of $H_{\mathcal{P}}(U)$, for each $i \in [m-1]$, there is a node $u_i \in U$ such that $\ell_i, \ell_{i+1} \in L(u_i)$. Therefore, for each $i \in [m]$, we have $L(u_{i-1}) \cap L(u_i) \neq \emptyset$, so there is an edge between u_{i-1} and u_i in $G_{\mathcal{P}}(U)$. Thus, the sequence $\langle u_0 = v, u_1, \ldots, u_m = w \rangle$ is a path between v and w in $G_{\mathcal{P}}(U)$. Hence, v and w are in the same connected component of $G_{\mathcal{P}}(U)$. □

3.4 Testing Compatibility

Here we present our tree compatibility algorithm. Section 3.4.1 gives an overview of and pseudocode for the algorithm. We prove the correctness of our algorithm in Section 3.4.2. In Section 3.4.3, we give an equivalent iterative version of the algorithm. The recursive and nonrecursive versions of our algorithm rely on the cluster intersection graph. In Section 3.4.4 we describe how to adapt our algorithms to use the display graph instead. As explained in Section 3.5, the iterative algorithm, combined with the display graph, provides an efficient solution to the tree compatibility problem.

From this point forward, we assume that $G_{\mathcal{P}}(U_{\text{init}})$ is connected. No generality is lost by making this assumption. To see why, observe that if $G_{\mathcal{P}}(U_{\text{init}})$ is not connected, then $\mathcal{P}$ can be partitioned into a

collection of species-disjoint profiles $\mathcal{P}_1, \ldots, \mathcal{P}_r$ such that $\mathcal{P}$ is compatible if and only if $\mathcal{P}_j$ is compatible, for all $j \in [r]$.

3.4.1 Overview of the Algorithm

Given a valid position U in $\mathcal{P}$ such that $G_{\mathcal{P}}(U)$ is connected, BUILDST(U) (Algorithm 2) determines whether or not $L(U)$ is compatible, and, if so, returns a phylogenetic tree T_U that displays U. BUILDST is closely related to the BUILD algorithm, reviewed in Section 3.3.1 (Algorithm 1). The key difference is that the latter uses the triple graph $\Gamma(\mathcal{P}|A)$, for different subsets A of $L(\mathcal{P})$, while the former uses the cluster intersection graph $G_{\mathcal{P}}(U)$, for different valid positions U in $\mathcal{P}$. This exploits the fact that, by Lemma 9, the two graphs offer essentially the same information.

The steps of BUILDST are analogous to those of BUILD, with $L(U)$ replacing $L(\mathcal{P})$ and references to connected components of $\Gamma(\mathcal{P})$ replaced by references to connected components of $G_{\mathcal{P}}(U)$. The most significant difference is the loop in lines 11–12, which replaces U by its successor position, replacing semi-universal nodes by their children. This is because the one-to-one relationship between connected components of $G_{\mathcal{P}}(U)$ and $\Gamma(P|L(U))$ implied by Lemma 9 only holds in the absence of semi-universal nodes.

3.4.2 Correctness

We need an auxiliary lemma.

Lemma 12. *Let U be a valid position in $\mathcal{P}$. If* BUILDST(U) *returns a tree T_U, then T_U is a phylogenetic tree such that $L(T_U) = L(U)$.*

Proof. We use induction on $|L(U)|$. If $|L(U)| = 1$ or $|L(U)| = 2$, the claim is trivially true, so suppose $|L(U)| > 2$. Let $W_1, \ldots, W_p$ be the connected components of $G_{\mathcal{P}}(U)$ in step 13. By Lemma 4, each W_j is a valid position. Then, by construction, the sets $L(W_1), \ldots, L(W_p)$ are pairwise disjoint and $L(U) = \bigcup_{j=1}^{p} L(W_j)$. Since BUILDST$(U)$ returns tree T_U, it must be the case that, for each $j \in [p]$, the result t_j returned by the recursive call to BUILDST(W_j) in step 17 is a tree, which we can assume inductively to be a phylogenetic tree for $L(W_j)$. Since $p \geq 2$, the tree with root r_U returned in step 22 is a phylogeny with species set $L(U)$. □

Algorithm 2: BUILDST(U)

Input: A valid position $U \subseteq V(\mathcal{P})$ such that $G_{\mathcal{P}}(U)$ is connected.

Output: A tree T_U that displays U, if U is compatible; `incompatible` otherwise.

```
1  Create a node r_U
2  if |L(U)| = 1 then
3  |  Let ℓ be the label in L(U)
4  |  return the tree consisting of node r_U, labeled by ℓ
5  if |L(U)| = 2 then
6  |  Let ℓ_1, ℓ_2 be the two labels in L(U)
7  |  foreach j ∈ [2] do
8  |  |  Create a node r_j, labeled ℓ_j
9  |  |  Make r_U the parent of r_j
10 |  return the tree with root r_U
   /* Compute the successor of U. */
11 foreach semi-universal node in v ∈ U do
12 |  U = (U \ {v}) ∪ Ch(v)
13 Let W_1, W_2, ..., W_p be the connected components of G_P(U)
14 if p = 1 then
15 |  return incompatible
16 foreach j ∈ [p] do
17 |  Let t_j = BUILDST(W_j)
18 |  if t_j is a tree then
19 |  |  Make r_U the parent of r(t_j)
20 |  else
21 |  |  return incompatible
22 return the tree with root r_U
```

We are now ready to prove the correctness of BUILDST.

Theorem 1. *Let U_{init} be the set defined in Equation (3.1). Then,* BUILDST(U_{init}) *either (i) returns a tree T that displays $\mathcal{P}$, if $\mathcal{P}$ is compatible, or (ii) returns* `incompatible` *otherwise.*

Proof. We first argue that if BUILDST(U_{init}) outputs `incompatible`, $\mathcal{P}$ is indeed incompatible. Assume, on the contrary, that $\mathcal{P}$ is compatible. Then, there must be a call BUILDST(U) for some valid position U, with $|L(U)| > 2$, such that BUILDST(U) returns `incompatible` in Line 15. By Lemma 6, U must be compatible. Let us consider the steps leading up to Line 15 during the execution of BUILDST(U).

By Lemma 10, immediately before Lines 11–12, U has a non-empty set of semi-universal nodes. By Lemma 7, after Lines 11–12, U is a valid position that contains no semi-universal nodes. Since Line 15

returns `incompatible`, it must be the case that $G_{\mathcal{P}}(U)$ is connected at this line. But then, by Lemma 10, U must be incompatible, a contradiction.

Now, suppose that BUILDST(U_{init}) returns a tree T. We prove that T displays $\mathcal{P}$ by arguing that for each $i \in [k]$ there is an injective mapping $\phi_i : V(T_i) \to V(T)$ that maps every node $v \in V(T_i)$ to a distinct node $\phi_i(v) \in V(T)$ such that $L(v) \subseteq L(\phi_i(v))$. This implies that $\text{Cl}(T_i) \subseteq \text{Cl}(T|L(T_i))$, for each $i \in [k]$; hence, T displays T_i.

By Lemma 12, each recursive call BUILDST(U) returns a phylogenetic tree T_U for $L(U)$. Let r_U denote the root of T_U. We have two cases.

Case (i): $|L(U)| \leq 2$. For each $i \in [k]$, we must have $|U(i)| \in \{0, 1, 2\}$; we only need to consider $|U(i)| \in \{1, 2\}$. Suppose first that $|U(i)| = 1$, and let v be the single node in $U(i)$. Note that $L(v) \subseteq L(r_U)$. Thus, we make $\phi_i(v) = r_U$. If $|L(U(i))| = 1$, we are done. Otherwise, $|L(U(i))| = 2$. Then, v has two children, v_1 and v_2, both leaves, labeled with, say, species s_1 and s_2, respectively. Node r_U also has two children, r_1 and r_2. Assume, without loss of generality, that these children are labeled with species s_1 and s_2, respectively. Then, $L(v_j) = L(r_j)$ for $j \in \{1, 2\}$. Therefore, we make $\phi_i(v_j) = r_j$ for each $j \in \{1, 2\}$. Now, suppose that $|U(i)| = 2$. Then, $|L(U(i))| = 2$, and each node in $U(i)$ is a leaf in T_i. As in the previous case, we map each node of $U(i)$ to the corresponding child of r_U.

Case (ii): $|L(U)| > 2$. Let $S(U)$ be the set of semi-universal nodes in U and let U' be the successor of U, computed in lines 11–12. Thus, $U' = (U \setminus S(U)) \cup \{u \in \text{Ch}(v) : v \in S(U)\}$. Assume inductively that for every strict descendant w of a node in U', there is a $j \in [p]$ such that w is mapped to a node in the tree t_j computed in Line 17. It therefore suffices to establish mappings for the nodes in $S(U)$. Now, for every $v \in S(U)$, $L(v) \subseteq L(r_U)$. Thus, we make $\phi(v) = r_U$ for every $v \in S(U)$. □

3.4.3 An Iterative Version

Although we described BUILDST as a recursive algorithm, we can also express it iteratively. As we will see in Section 3.5, the iterative version lends itself naturally to an implementation.

Algorithm 3, BUILDST$_{\text{N}}$ (where the "N" stands for "non-recursive"), performs a breadth-first traversal of BUILDST's recursion tree using a first-in first-out queue Q. This queue stores pairs of the form (U, pred),

Algorithm 3: BUILDST$_{\mathrm{N}}(\mathcal{P})$

Input: A profile $\mathcal{P}$.

Output: A tree T that displays $\mathcal{P}$, if $\mathcal{P}$ is compatible; `incompatible` otherwise.

```
1  Construct G_P(U_init)
2  ENQUEUE(Q, (U_init, null))
3  while Q is not empty do
4      (U, pred) = DEQUEUE(Q)
5      Create a node r_U and set parent(r_U) = pred
6      if |L(U)| = 1 then
7          Let ℓ be the label in L(U)
8          Label r_U with ℓ
9          continue
10     if |L(U)| = 2 then
11         Let ℓ_1, ℓ_2 be the two labels in L(U)
12         foreach j ∈ [2] do
13             Create a node r_j, labeled ℓ_j
14             Set parent(r_j) = r_U
15         continue
       /* Compute the successor of U. */
16     foreach semi-universal node in v ∈ U do
17         U = (U \ {v}) ∪ Ch(v)
18     Let W_1, W_2, ..., W_p be the connected components of G_P(U)
19     if p = 1 then
20         return incompatible
21     foreach j ∈ [p] do
22         ENQUEUE(Q, (W_j, r_U))
23 return the tree with root r_{U_init}
```

where U is a valid position in $\mathcal{P}$ and pred is a reference to the parent of the node corresponding to U in the supertree built so far.

BUILDST$_{\mathrm{N}}$ initializes its queue to contain the starting position, U_{init}, with a null parent. It then proceeds to the **while** loop of Lines 3–22. Each iteration of the loop starts by dequeuing a valid position U, along with a reference pred to the potential parent for the subtree for $L(U)$ in the supertree. BUILDST$_{\mathrm{N}}$ then creates a tentative root r_U for the tree T_U for $L(U)$, and links r_U to its parent. In Lines 6–15, it considers the trivial cases $|L(U)| = 1$ or 2, in a manner analogous to that of BUILDST. Here, the **continue** statement indicates

that the rest of the current iteration of the **while** loop should be skipped, and the algorithm should continue to the next iteration.

After considering the trivial cases, BUILDST$_{\mathrm{N}}$ proceeds to in Lines 16–18, where — in the same manner as BUILDST — it computes the successor of U and recomputes the connected components of $G_{\mathcal{P}}(U)$. If $G_{\mathcal{P}}(U)$ has only one component, then, as argued in the proof of Theorem 1, $\mathcal{P}$ must be incompatible, which is reported in Line 20. Otherwise, Lines 21–22 enqueue each of the connected components $W_1, \ldots, W_p$ of $G_{\mathcal{P}}(U)$, along with r_U. Each W_j is processed in a subsequent iteration, in which the root for the subtree for $L(W_j)$, if such a tree exists, will be linked to its parent r_U. If the **while** loop terminates without any incompatibility being detected, the algorithm returns the tree with root $r_{U_{\mathrm{init}}}$.

Note that the order in which BUILDST$_{\mathrm{N}}$ processes connected components differs from that of BUILDST: the former proceeds breadth-first, while the latter does so depth-first. Nevertheless, it is straightforward to see that the effect is equivalent, and the proof of correctness of BUILDST (Theorem 1) applies to BUILDST$_{\mathrm{N}}$ as well. We thus state the following result without proof.

Theorem 2. *Let $\mathcal{P} = \{T_1, \ldots, T_k\}$ be a profile. Then,* BUILDST$_{\mathrm{N}}(\mathcal{P})$ *either (i) returns a tree T that displays $\mathcal{P}$, if $\mathcal{P}$ is compatible, or (ii) returns* `incompatible` *otherwise.*

3.4.4 Using the Display Graph

There is a potential difficulty in maintaining the connected components of graph $G_{\mathcal{P}}(U)$, as required by BUILDST and BUILDST$_{\mathrm{N}}$: the edges of $G_{\mathcal{P}}(U)$ are defined via set intersections, which can make it costly to update $G_{\mathcal{P}}(U)$ after computing the successor of U (e.g., as in Lines 16–17 of Algorithm 3). We can circumvent this difficulty by using the graph $H_{\mathcal{P}}(U)$ of Section 3.3.3 as a proxy for $G_{\mathcal{P}}(U)$ in our algorithms. Next, we explain how to do so, focusing on the iterative version, BUILDST$_{\mathrm{N}}$.

By Lemma 11(ii), the connected components $W_1, \ldots, W_p$ of $G_{\mathcal{P}}(U)$ can be put into a one-to-one correspondence with the connected components $Y_1, \ldots, Y_p$ of $H_{\mathcal{P}}(U)$ so that $W_j = Y_j \cap U$ for each $j \in [p]$. Thus, $H_{\mathcal{P}}(U)$ offers the same connectivity information as $G_{\mathcal{P}}(U)$. On the other hand, offers an important advantage over $G_{\mathcal{P}}(U)$: maintaining the connected components of $H_{\mathcal{P}}(U)$ only requires performing edge and vertex deletions; there is no need to recompute set intersections. Indeed, by Lemma 11(i), we can per-

form Lines 16–17 of BUILDST$_{\mathrm{N}}$ by deleting each edge (v, u) such that $u \in \mathrm{Ch}(v)$ from $H_{\mathcal{P}}(U)$, and then deleting v from $H_{\mathcal{P}}(U)$.

As BUILDST$_{\mathrm{N}}(\mathcal{P})$ is executed, and edges and vertices of $H_{\mathcal{P}} = H_{\mathcal{P}}(U_{\mathrm{init}})$ are deleted, the graph, which is initially connected, becomes increasingly fragmented. Let H_{cur} be the subgraph of $H_{\mathcal{P}}(U_{\mathrm{init}})$ that remains at the beginning of the current iteration of BUILDST$_{\mathrm{N}}$'s **while** loop. Each entry of Q now corresponds to a distinct component Y of H_{cur} such that Y is the set of vertices of $H_{\mathcal{P}}(U)$ for some valid position U. It is easy to see that U consists of those nodes in $Y \setminus X_{\mathcal{P}}$ that have no parent in Y. It is also clear that the total size of all the components stored in Q at any point during the execution of BUILDST$_{\mathrm{N}}$ is $O(M_{\mathcal{P}})$. As we explain in the next section, we can maintain the connected components in $O(\log^2 M_{\mathcal{P}})$ time per edge or node deletion. Since BUILDST$_{\mathrm{N}}$ performs $O(M_{\mathcal{P}})$ deletions, the total time to maintain the components is $O(M_{\mathcal{P}} \log^2 M_{\mathcal{P}})$. In addition to connectivity information, we also need a way to quickly identify semi-universal nodes, and to perform other bookkeeping operations. We shall see that this additional work also takes $O(M_{\mathcal{P}} \log^2 M_{\mathcal{P}})$ time.

3.5 Time Complexity

The main result of this section is the following.

Theorem 3. *There is an algorithm that, given a profile $\mathcal{P}$ of rooted phylogenetic trees, runs in $O(M_{\mathcal{P}} \log^2 M_{\mathcal{P}})$ time, and either returns a tree that displays $\mathcal{P}$, if $\mathcal{P}$ is compatible, or reports that is $\mathcal{P}$ is incompatible otherwise.*

We prove Theorem 3 by showing how to implement BUILDST$_{\mathrm{N}}$ so that BUILDST$_{\mathrm{N}}(\mathcal{P})$ runs in $O(M_{\mathcal{P}} \log^2 M_{\mathcal{P}})$ time. We assume that the implementation of BUILDST$_{\mathrm{N}}$ uses $H_{\mathcal{P}}(U)$ instead of $G_{\mathcal{P}}(U)$, as explained in Section 3.4.4. The key is to maintain the following three pieces of information for every valid position U that is processed in the **while** loop of Lines 3–22.

(I1) *The value of $|L(U)|$ and, if $|L(U)| \in \{1, 2\}$, the species in $L(U)$.* This information is needed to determine if U falls into one of the trivial cases handled in Lines 6–15, and to handle these cases, if they apply.

(I2) *The set $S(U)$ of semi-universal nodes in U.* This set is used to compute the successor of U in Lines 16–17.

(I3) *The connected components of $H_{\mathcal{P}}(U)$.* Although $H_{\mathcal{P}}(U)$ is connected at the beginning of each iteration, the graph may break into multiple connected components $Y_1, \ldots, Y_p$ after Lines 16–17. The information about these components is needed in Lines 18–22. We also need to compute $|L(Y_j \cap U)|$ and $S(Y_j \cap U)$, for each $j \in [p]$, for use in subsequent iterations.

In the rest of this section, we describe the data structures used by BUILDST$_{\mathrm{N}}$, and how these data structures are initialized and then maintained throughout the execution. Finally, we analyze the running time of our algorithm.

3.5.1 Data Structures

We represent $H_{\mathcal{P}}(U)$ using the dynamic graph connectivity data structure of Holm et al. (28), which we refer to as *HDT*. HDT maintains a graph G under a series of updates — edge insertions or deletions — interspersed with queries asking whether two given nodes are in the same connected component. If we start with no edges in a graph with N vertices, HDT guarantees that the amortized cost of each operation is $O(\log^2 N)$. HDT can be implemented using $O(N)$ space (24). BUILDST$_{\mathrm{N}}$ requires three operations that are not explicitly specified by Holm et al. (28):

1. detecting whether the deletion of an edge splits a connected component of G in two,
2. determining the number of nodes in a connected component of G, and
3. iterating through the nodes of a connected component of G.

The need for operation 1 is obvious. The need for operations 2 and 3 will become clear later.

Let N be the number of nodes in the graph G and m be the number of nodes in the connected component under consideration. We now explain how to adapt HDT so that operations 1, 2, and 3 take, respectively, $O(\log^2 N)$ amortized time, $O(\log N)$ time, and $O(m)$ time (where the latter does not count the time to process each node).

Implementing operation 1 is straightforward. Let $e = (u, v)$ be the edge to be deleted. Then, to see if the deletion of e breaks the graph into two components, we simply delete e and then test if u and v are in the same component. This requires two operations on HDT, for a total of $O(\log^2 N)$ amortized time.

To explain how to implement operations 2 and 3, we first need to review the main features of HDT. Our focus is on the aspects that are crucial to our algorithm. For a full description of HDT, we refer the reader to the original reference (28).

HDT maintains a spanning forest of the graph G, where each tree in the forest is a spanning tree for a connected component. An edge in the graph is a *tree edge* if it belongs to some spanning tree; otherwise it is a *non-tree edge*. Each spanning tree is represented using an *Euler Tour tree* (*ET-tree*), a dynamic balanced search tree that represents an Euler tour of the tree. An Euler tour of a tree S is obtained by replacing every edge in S be two edges in opposite directions, and then finding a cycle in the resulting graph that uses each edge exactly once. Thus, a spanning tree with m nodes is represented by an ET tree with $2m - 2$ nodes. A node in G may contribute multiple vertices to the ET tree that contains it. Any one of these tree vertices is chosen as the *representative* of that node. We note that, in fact, HDT maintains multiple ET trees for each component, one of which is the aforementioned ET tree for the spanning tree of the component. Those additional ET trees are essential for achieving the time bounds proved by Holm et al., but they have no impact on our analysis. Thus, we will not consider those trees in the subsequent discussion.

Given any vertex v in an ET tree, one can determine in $O(\log N)$ time the number of vertices in the tree containing v. From this number, we can easily obtain the number of nodes in the connected component containing v in $O(1)$ time. Hence, we can implement operation 2 to run in $O(\log N)$ time.

To iterate through the nodes in a connected component of G, we traverse the ET tree for the component, ignoring any vertex that is not a representative for a node in the component. Since the number of vertices in the ET tree for an m-node connected component is $O(m)$, iterating through the nodes in the connected component, as required for operation 3, takes $O(m)$ time (aside from the time spent processing the node).

Observe that the total number of edge and vertex deletions performed by $\textsc{BuildST}_{\mathrm{N}}(\mathcal{P})$ cannot exceed the total number of edges and vertices in $H_{\mathcal{P}}$, which is $O(M_{\mathcal{P}})$. Each update on $H_{\mathcal{P}}(U)$ takes $O(\log^2 M_{\mathcal{P}})$ amortized time. Therefore, the HDT data structure allows us to maintain connectivity information through-

out the entire algorithm in $O(M_{\mathcal{P}} \log^2 M_{\mathcal{P}})$ time, which is within the time bound claimed in Theorem 3. Thus, from this point forward, we focus on how to maintain (I1) and (I2).

Let H_{cur} be the subgraph of $H_{\mathcal{P}}(U_{\text{init}})$ that remains at the beginning of the current iteration of BUILDST$_{\text{N}}$'s **while** loop. We assume that each node v in H_{cur} has the following two pieces of information.

- An integer $v.\texttt{index} \in [k+1]$. If $v \in X_{\mathcal{P}}$, then $v.\texttt{index} = k+1$; otherwise $v.\texttt{index}$ is the index $i \in [k]$ such that $v \in V(T_i)$. The value of $v.\texttt{index}$ is fixed throughout the execution of BUILDST$_{\text{N}}$.

- A Boolean flag $v.\texttt{mark}$. If $v.\texttt{mark}$ is `true`, we say that v is *marked*; otherwise, v is *unmarked*. The value of $v.\texttt{mark}$ is maintained so that, at the beginning of each iteration of BUILDST$_{\text{N}}$'s **while** loop, v is marked if and only $v.\texttt{index} \neq k+1$ and v has no parent in H_{cur}. Thus, if v is in, say, connected component Y of H_{cur}, then $v \in U$ for some valid position U such that the vertex set of $H_{\mathcal{P}}(U)$ is precisely Y.

For each connected component Y of H_{cur}, we maintain three fields:

- $Y.\texttt{count}$, the cardinality of $Y \cap X_{\mathcal{P}}$, where $X_{\mathcal{P}}$ is the special set of labeled nodes defined in Equation (3.2),

- $Y.\texttt{map}$, a map defined as follows. For each $i \in [k]$, let $L_i = \{v \in Y : v.\texttt{index} = i \text{ and } v.\texttt{mark} = \texttt{true}\}$. Then, $Y.\texttt{map}$ consists of all pairs (i, L_i), such that $i \in [k]$ and $L_i \neq \emptyset$. For each $i \in [k]$ such that $L_i \neq \emptyset$, $Y.\texttt{map}(i)$ denotes the set L_i.

- $Y.\texttt{semiU}$, a set consisting of all $i \in [k]$ such that $Y.\texttt{map}(i)$ is defined, $|Y.\texttt{map}(i)| = 1$, and the single element of $Y.\texttt{map}(i)$ is an internal node in T_i.

We assume that `map` fields are implemented using balanced binary search trees. Thus, given any index $i \in [k]$, we can, in $O(\log M_{\mathcal{P}})$ time per operation, access $Y.\texttt{map}(i)$ or determine that $Y.\texttt{map}(i)$ is undefined. (Note that the stated $O(\log M_{\mathcal{P}})$ time bound per operation is an overestimate — the actual time is $O(\log k)$ — but the overestimate suffices for our analyses.) We can also add or delete an entry (i, L_i) to $Y.\texttt{map}$ in $O(\log M_{\mathcal{P}})$ time. The set $Y.\texttt{map}(i)$ is itself implemented using a balanced binary search tree. We also

assume that the `semiU` fields are implemented using balanced binary search trees, so that access and updates (insertions and deletions) can be performed in $O(\log M_{\mathcal{P}})$ time per operation.

Suppose Y is the vertex set of $H_{\mathcal{P}}(U)$ for the valid position U extracted from Q at the beginning of an iteration of the **while** loop of Lines 3–22. Then, $|L(U)| = Y.\texttt{count}$ and $S(U) = \{v \in Y.\texttt{map}(i) : i \in Y.\texttt{semiU}\}$. Moreover, if $|L(U)| \in \{1, 2\}$, we can easily identify the labels in $L(U)$, by examining the species labels of the nodes $v \in Y$ such that $v.\texttt{index}$ is $k + 1$. Thus, the data fields we have defined provide all the information needed by each iteration of BUILDST$_{\text{N}}$'s **while** loop.

In the next subsections, we describe how to initialize all the required data fields for $H_{\mathcal{P}} = H_{\mathcal{P}}(U_{\text{init}})$ and maintain the data fields for each connected component of the current graph H_{cur} after each edge and vertex deletion.

3.5.2 Initializing the Data Fields

Graph $H_{\mathcal{P}} = H_{\mathcal{P}}(U_{\text{init}})$ has a single connected component $Y_{\text{init}} = V(\mathcal{P}) \cup X_{\mathcal{P}}$, which is the entire vertex set of the graph. Consider any node v in $H_{\mathcal{P}}$. It is straightforward to initialize $v.\texttt{index}$. We set $v.\texttt{mark} = \texttt{true}$ if $v = r(T_i)$ for some $i \in [k]$; otherwise, $v.\texttt{mark} = \texttt{false}$.

We initialize the data fields of Y_{init} as follows.

- $Y_{\text{init}}.\texttt{count} = |L(\mathcal{P})|$.
- $Y_{\text{init}}.\texttt{map}$ consists of all pairs $(i, \{r(T_i)\})$, for each $i \in [k]$.
- $Y_{\text{init}}.\texttt{semiU} = [k]$.

It is straightforward to initialize all data fields in $O(M_{\mathcal{P}})$ time.

3.5.3 Maintaining the Data Fields

Suppose that the `mark`, `count`, `semiU`, and `map` fields are correctly computed for every vertex and connected component that exists at the beginning of an iteration of the **while** loop in 3–22 of BUILDST$_{\text{N}}$. We now show how to maintain the `mark`, `count`, `semiU`, and `map` fields for the new connected components created by Lines 16–17, in such a way that the total time spent in this process throughout the entire execution

of BUILDST$_{\mathrm{N}}(\mathcal{P})$ is $O(M_{\mathcal{P}} \log^2 M_{\mathcal{P}})$. We assume, conservatively, that the time to perform a single update on any data field is $O(\log M_{\mathcal{P}})$.

3.5.3.1 Computing Successor Positions

To implement the loop in lines 16–17, we first identify the set of semi-universal nodes. As explained earlier, this set is given by $S(U) = \{v \in Y.\texttt{map}(i) : i \in Y.\texttt{semiU}\}$. Next, we make $Y.\texttt{semiU} = \emptyset$. We then successively process each node v in $S(U)$ as follows. Let $i = v.\texttt{index}$; note that we can assume that $i \neq k+1$. We remove $(i, \{v\})$ from $Y.\texttt{map}$. Then, we add $(i, \mathrm{Ch}(v))$ to $Y.\texttt{map}$. We unmark v, and, for every node $u \in \mathrm{Ch}(v)$, we mark u. We then successively delete each edge (v, u) such that $u \in \mathrm{Ch}(v)$, updating the `count`, `semiU`, and `map` fields for each newly-created component (we explain the details in the next subsection). Once these edges are deleted, we delete v itself from Y.

The total number of data field updates needed to perform the above operations for each node v, aside from the edge deletions, is $O(1 + |\mathrm{Ch}(v)|)$. The total time over the entire execution of BUILDST$_{\mathrm{N}}(\mathcal{P})$ is therefore $O(M_{\mathcal{P}} \log M_{\mathcal{P}})$.

Next let us focus on how to handle the deletion of a single edge in Case 2.

3.5.3.2 Deleting an Edge

Let $e = (v, u)$ be an edge to be deleted in Case 2 above; assume that $v \in V(T_i)$. Let Y' be the connected component of H_{cur} that contains v. We query the HDT data structure to determine, in $O(\log^2 M_{\mathcal{P}})$ amortized time, whether deleting (v, u) splits Y' into two components.

If Y' remains connected, no more updates are needed. Otherwise, Y' is split into two subcomponents Y_1 and Y_2, containing v and u, respectively. To fill in the `count`, `semiU`, and `map` fields of Y_1 and Y_2, we use the well-known technique of scanning the smaller component (10). We query HDT to determine which of Y_1 and Y_2 has fewer nodes. Suppose without loss of generality that $|Y_1| \leq |Y_2|$. We initialize $Y_2.\texttt{count}$, $Y_2.\texttt{semiU}$, and $Y_2.\texttt{map}$ to $Y'.\texttt{count}$, $Y'.\texttt{semiU}$, and $Y'.\texttt{map}$, respectively. We initialize $Y_1.\texttt{count}$ to 0, $Y_1.\texttt{semiU} = \emptyset$, and $Y_1.\texttt{map} = \emptyset$. We then iterate through each node v in Y_1, and do the following. If $v \in X_{\mathcal{P}}$, we decrement $Y_2.\texttt{count}$ and increment $Y_1.\texttt{count}$. Otherwise $v \in V(T_i)$, where $i = v.\texttt{index}$.

If v is marked, we remove v from $Y_2.\texttt{map}(i)$ and add v to $Y_1.\texttt{map}(i)$. As we do this, for each $j \in \{1, 2\}$, we check whether, it is necessary to add or remove i from $Y_j.\texttt{semiU}$. This decision depends on whether, as a result of moving v, we have that $|Y_j.\texttt{map}(i)| = 1$ and the single element of $Y.\texttt{map}(i)$ is an internal node in T_i. The test to determine this can be performed in $O(\log M_{\mathcal{P}})$ time. Note that each update on a `map` or `semiU` field takes $O(\log M_{\mathcal{P}})$ time.

We claim that any node v is scanned $O(\log M_{\mathcal{P}})$ times over the entire execution of BUILDST$_{\text{N}}(\mathcal{P})$. To verify this, let $N(v)$ be the number of nodes in the connected component containing v. Suppose that, initially, $N(v) = N$. Then, the rth time we scan v, $N(v) \leq N/2^r$. Thus, v is scanned $O(\log N)$ times. The claim follows, since $N = O(M_{\mathcal{P}})$. Therefore, the total number of updates over all nodes is $O(M_{\mathcal{P}} \log M_{\mathcal{P}})$, for a total time of $O(M_{\mathcal{P}} \log^2 M_{\mathcal{P}})$.

3.5.4 Summary

Let us review the running times of each aspect of our implementation of BUILDST$_{\text{N}}$.

1. **Initializing the data structures.** This step has two parts.
   - *Setting up the HDT data structure for $H_{\mathcal{P}}$.* This takes $O(M_{\mathcal{P}} \log^2 M_{\mathcal{P}})$ time.
   - *Initializing the data fields for the single connected component of $H_{\mathcal{P}}$.* This takes $O(M_{\mathcal{P}})$ time.
2. **Maintaining the data structures.** This step also has two parts.
   - *Updating the HDT data structure*. There are $O(M_{\mathcal{P}})$ edge and node deletions, at an amortized cost of $O(\log^2 M_{\mathcal{P}})$ per deletion, yielding a total time of $O(M_{\mathcal{P}} \log^2 M_{\mathcal{P}})$.
   - *Maintaining the relevant data fields for the connected components.* This takes a total of $O(M_{\mathcal{P}} \log^2 M_{\mathcal{P}})$ over the entire execution of BUILDST.

We conclude that the total running time of BUILDST$_{\text{N}}(\mathcal{P})$ is $O(M_{\mathcal{P}} \log^2 M_{\mathcal{P}})$, completing the proof of Theorem 3.

3.6 Discussion

A trivial lower bound for the tree compatibility problem is $\Omega(M_{\mathcal{P}})$, the time to read the input. Thus, our result leaves us a polylogarithmic factor away from an optimal algorithm for compatibility. Is it possible to reduce or even eliminate this gap? The bottleneck is the time to maintain the information associated with the various components of $H_{\mathcal{P}}(U)$. It is conceivable that the special structure of this graph and the way the deletions are performed could be used to our advantage. A second question is how well our algorithm performs in practice. To investigate this, it should be possible to leverage existing knowledge on the empirical behavior of dynamic connectivity data structures (16).

In recent work (8), we have extended the approach presented here to develop a $O(M_{\mathcal{P}} \log^2 M_{\mathcal{P}})$ algorithm to test the compatibility of profiles of *semi-labeled trees*, that is, phylogenies whose internal nodes may be labeled by higher-order taxa. This extension enables us incorporate *taxonomies* (that is, trees that group organisms according to a system of taxonomic rank — e.g., family, genus, and species) as input trees. The use of taxonomies can broaden the taxonomic coverage of supertree analyses significantly (14).

CHAPTER 4. NESTED TAXA TREES COMPATIBILITY CHECKING

4.1 Our Contributions

The $\tilde{O}(M_{\mathcal{P}})$ running time of our ancestral compatibility algorithm is independent of the degrees of the nodes of the input trees, a valuable characteristic for large datasets that include taxonomies. To achieve this time bound, we extend ideas from our recent algorithm for testing the compatibility of ordinary phylogenetic trees (9). As in that algorithm, a central notion in the current paper is the *display graph* of profile $\mathcal{P}$, denoted $H_{\mathcal{P}}$. This is the graph obtained from the disjoint union of the trees in $\mathcal{P}$ by identifying nodes that have the same label (see Section 4.4). The term "display graph" was introduced by Bryant and Lagergren (4), but similar ideas have been used elsewhere. In particular, the display graph is closely related to Berry and Semple's *restricted descendancy graph* (25), a mixed graph whose directed edges correspond to the (undirected) edges of $H_{\mathcal{P}}$ and whose undirected edges have no correspondence in $H_{\mathcal{P}}$. The second kind of edges are the major component of the $\tau_{\mathcal{P}}$ term in the time and space complexity of Berry and Semple's algorithm. The absence of such edges makes $H_{\mathcal{P}}$ significantly smaller than the restricted descendancy graph. Display graphs also bear some relation to *tree alignment graphs* (31).

Here, we exploit the display graph more extensively and more directly than our previous work. Although the display graph of a collection of semi-labeled trees is more complex than that of a collection of ordinary phylogenies, we are able to extend several of the key ideas — notably, that of a semi-universal label — to the general setting of semi-labeled trees. As in (9), the implementation relies on a dynamic graph data structure, but it requires a more careful amortized analysis based on a weighing scheme.

4.2 Preliminaries

4.2.1 Semi-Labeled Trees

A *semi-labeled tree* is a pair $\mathcal{T} = (T, \phi)$ where T is a tree and ϕ is a mapping from a set $L(\mathcal{T})$ to $V(T)$ such that, for every node $v \in V(T)$ of degree at most two, $v \in \phi(L(\mathcal{T}))$. $L(\mathcal{T})$ is the *label set* of $\mathcal{T}$ and ϕ is the *labeling function* of $\mathcal{T}$.

For every node $v \in V(T)$, $\phi^{-1}(v)$ denotes the (possibly empty) subset of $L(\mathcal{T})$ whose elements map into v; these elements as the *labels of* v (thus, each label is a taxon). If $\phi^{-1}(v) \neq \emptyset$, then v is *labeled*; otherwise, v is *unlabeled*. Note that, by definition, every leaf in a semi-labeled tree is labeled. Further, any node, including the root, that has a single child must be labeled. Nodes with two or more children may be labeled or unlabeled. A semi-labeled tree $\mathcal{T} = (T, \phi)$ is *singularly labeled* if every node in T has at most one label; $\mathcal{T}$ is *fully labeled* if every node in T is labeled.

Semi-labeled trees, also known as *X-trees*, generalize ordinary phylogenetic trees, also known as *phylogenetic X-trees* (21). An ordinary phylogenetic tree is a semi-labeled tree $\mathcal{T} = (T, \phi)$ where $r(T)$ has degree at least two and ϕ is a bijection from $L(\mathcal{T})$ into leaf set of T (thus, internal nodes are not labeled).

Let $\mathcal{T} = (T, \phi)$ be a semi-labeled tree and let ℓ and ℓ' be two labels in $L(\mathcal{T})$. If $\phi(\ell) \leq_T \phi(\ell')$, then we write $\ell \leq_{\mathcal{T}} \ell'$, and say that ℓ' is a *descendant* of ℓ in $\mathcal{T}$ and that ℓ is an *ancestor* of ℓ'. We write $\ell <_{\mathcal{T}} \ell'$ if $\phi(\ell')$ is a proper descendant of $\phi(\ell)$. If $\phi(\ell) \parallel_T \phi(\ell')$, then we write $\ell \parallel_{\mathcal{T}} \ell'$ and say that ℓ and ℓ' are *not comparable* in $\mathcal{T}$. If $\mathcal{T}$ is fully labeled and $\phi(\ell)$ is the parent of $\phi(\ell')$ in T, then ℓ is the *parent* of ℓ' in $\mathcal{T}$ and ℓ' is a *child* of ℓ in $\mathcal{T}$; two labels with the same parent are *siblings*.

Two semi-labelled trees $\mathcal{T} = (T, \phi)$ and $\mathcal{T}' = (T', \phi')$ are *isomorphic* if there exists a bijection $\psi : V(T) \to V(T')$ such that $\phi' = \psi \circ \phi$ and, for any two nodes $u, v \in V(T)$, $(u, v) \in E(T)$ if and only $(\psi(u), \psi(v)) \in E(T')$.

Let $\mathcal{T} = (T, \phi)$ be a semi-labeled tree. For each $u \in V(T)$, $X(u)$ denotes the set of all labels in the subtree of T rooted at u; that is, $X(u) = \bigcup_{v:u \leq_T v} \phi^{-1}(v)$. $X(u)$ is called a *cluster* of T. $\mathrm{Cl}(\mathcal{T})$ denotes the set of all clusters of $\mathcal{T}$. It is well known (21, Theorem 3.5.2) that a semi-labeled tree $\mathcal{T}$ is completely

determined by $\mathrm{Cl}(\mathcal{T})$. That is, if $\mathrm{Cl}(\mathcal{T}) = \mathrm{Cl}(\mathcal{T}')$ for some other semi-labeled tree $\mathcal{T}'$, then $\mathcal{T}$ is isomorphic to $\mathcal{T}'$.

Suppose $A \subseteq L(\mathcal{T})$ for a semi-labeled tree $\mathcal{T} = (T, \phi)$. The *restriction* of $\mathcal{T}$ to A, denoted $\mathcal{T}|A$, is the semi-labeled tree whose cluster set is $\mathrm{Cl}(\mathcal{T}|A) = \{X \cap A : X \in \mathrm{Cl}(\mathcal{T}) \text{ and } X \cap A \neq \emptyset\}$. Intuitively, $\mathcal{T}|A$ is obtained from the minimal rooted subtree of T that connects the nodes in $\phi(A)$ by suppressing all vertices of degree two that are not in $\phi(A)$.

Let $\mathcal{T} = (T, \phi)$ and $\mathcal{T}' = (T', \phi')$ be semi-labeled trees such that $L(\mathcal{T}') \subseteq L(\mathcal{T})$. $\mathcal{T}$ *ancestrally displays* $\mathcal{T}'$ if $\mathrm{Cl}(\mathcal{T}') \subseteq \mathrm{Cl}(\mathcal{T}|L(\mathcal{T}'))$. Equivalently, $\mathcal{T}$ ancestrally displays $\mathcal{T}'$ if $\mathcal{T}'$ can be obtained from $\mathcal{T}|L(\mathcal{T}')$ by contracting edges, and, for any $\ell_1, \ell_2 \in L(\mathcal{T}')$,

(i) if $\ell_1 <_{\mathcal{T}'} \ell_2$, then $\ell_1 <_{\mathcal{T}} \ell_2$, and

(ii) if $\ell_1 \parallel_{\mathcal{T}'} \ell_2$, then $\ell_1 \parallel_{\mathcal{T}} \ell_2$.

The notion of "ancestrally displays" for semi-labeled trees generalizes the well-known notion of "displays" for ordinary phylogenetic trees (21).

For a semi-labelled tree $\mathcal{T}$, let us define $D(\mathcal{T})$ and $N(\mathcal{T})$ as follows.

$$D(\mathcal{T}) = \{(\ell, \ell') : \ell, \ell' \in L(\mathcal{T}) \text{ and } \ell <_{\mathcal{T}} \ell'\}$$
$$N(\mathcal{T}) = \{\{\ell, \ell'\} : \ell, \ell' \in L(\mathcal{T}) \text{ and } \ell \parallel_{\mathcal{T}} \ell'\}$$

Note that $D(\mathcal{T})$ consists of *ordered* pairs, while $N(\mathcal{T})$ consists of *unordered* pairs.

Lemma 13 (Bordewich et al. (26))**.** *Let $\mathcal{T}$ and $\mathcal{T}'$ be semi-labelled trees such that $L(\mathcal{T}') \subseteq L(\mathcal{T})$. Then $\mathcal{T}$ ancestrally displays $\mathcal{T}'$ if and only if $D(\mathcal{T}') \subseteq D(\mathcal{T})$ and $N(\mathcal{T}') \subseteq N(\mathcal{T})$.*

4.2.2 Profiles and Ancestral Compatibility

Throughout the rest of this paper $\mathcal{P} = \{\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_k\}$ denotes a set where, for each $i \in [k]$, $\mathcal{T}_i = (T_i, \phi_i)$ is a semi-labeled tree. We refer to $\mathcal{P}$ as a *profile*, and write $L(\mathcal{P})$ to denote $\bigcup_{i \in [k]} L(\mathcal{T}_i)$, the *label set* of $\mathcal{P}$. Figure 4.1 shows a profile where $L(\mathcal{P}) = \{a, b, c, d, e, f, g, h, i\}$. We write $V(\mathcal{P})$ for $\bigcup_{i \in [k]} V(T_i)$ and $E(\mathcal{P})$ for $\bigcup_{i \in [k]} E(T_i)$, The *size* of $\mathcal{P}$ is $M_{\mathcal{P}} = |V(\mathcal{P})| + |E(\mathcal{P})|$.

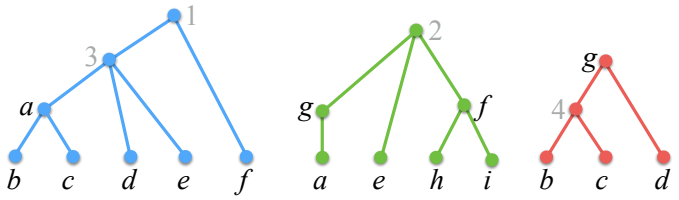

Figure 4.1 A profile $\mathcal{P} = \{\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3\}$ — trees are ordered left-to-right. The letters are the original labels; grey numbers are labels added to make the trees fully labeled. (Adapted from (25).)

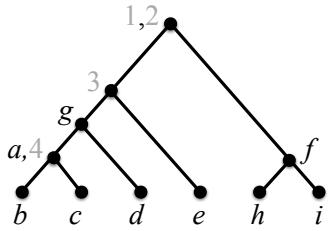

Figure 4.2 A tree $\mathcal{T}$ that ancestrally displays the profile of Figure 4.1. (Adapted from (25).)

$\mathcal{P}$ is *ancestrally compatible* if there is a rooted semi-labeled tree $\mathcal{T}$ that ancestrally displays each of the trees in $\mathcal{P}$. If $\mathcal{T}$ exists, we say that $\mathcal{T}$ *ancestrally displays* $\mathcal{P}$ (see Figure 4.2).

Given a subset X of $L(\mathcal{P})$, the *restriction* of $\mathcal{P}$ to X, denoted $\mathcal{P}|X$, is the profile defined as

$$\mathcal{P}|X = \{\mathcal{T}_1|X \cap L(\mathcal{T}_1), \mathcal{T}_2|X \cap L(\mathcal{T}_2), \ldots, \mathcal{T}_k|X \cap L(\mathcal{T}_k)\}.$$

The proof of the following lemma is straightforward.

Lemma 14. *Suppose $\mathcal{P}$ is ancestrally compatible and let $\mathcal{T}$ be a tree that ancestrally displays $\mathcal{P}$. Then, for any $X \subseteq L(\mathcal{P})$, $\mathcal{T}|X$ ancestrally displays $\mathcal{P}|X$.*

A semi-labeled tree $\mathcal{T} = (T, \phi)$ is *fully labeled* if every node in T is labeled. Suppose $\mathcal{P}$ contains trees that are not fully labeled. We can convert $\mathcal{P}$ into an equivalent profile $\mathcal{P}'$ of fully-labeled trees as follows. For each $i \in [k]$, let l_i be the number of unlabeled nodes in T_i. Create a set L' of $n' = \sum_{i \in [k]} l_i$ labels such that $L' \cap L(\mathcal{P}) = \emptyset$. For each $i \in [k]$ and each $v \in V(T_i)$ such that $\phi_i^{-1}(v) = \emptyset$, make $\phi_i^{-1}(v) = \{\ell\}$, where ℓ is a distinct element from L'. We refer to $\mathcal{P}'$ as the *profile obtained by adding distinct new labels to* $\mathcal{P}$ (see Figure 4.1).

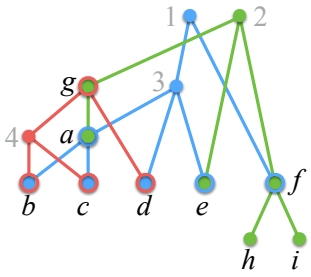

Figure 4.3 The display graph $H_{\mathcal{P}}$ for the profile of Figure 4.1.

Lemma 15 (Daniel and Semple (27))**.** *Let $\mathcal{P}'$ be the profile obtained by adding distinct new labels to $\mathcal{P}$. Then, $\mathcal{P}$ is ancestrally compatible if and only if $\mathcal{P}'$ is ancestrally compatible. Further, if $\mathcal{T}$ is a semi-labeled phylogenetic tree that ancestrally displays $\mathcal{P}'$, then $\mathcal{T}$ ancestrally displays $\mathcal{P}$.*

From this point forward, we make the following assumption.

Assumption 1. *For each $i \in [k]$, $\mathcal{T}_i$ is fully and singularly labeled.*

By Lemma 15, no generality is lost in assuming that all trees in $\mathcal{P}$ are fully labeled. The assumption that the trees are singularly labeled is inessential; it is only for clarity. Note that, even with the latter assumption, a tree that ancestrally displays $\mathcal{P}$ is not necessarily singularly labeled. Figure 4.2 illustrates this fact.

4.3 The Display Graph

The *display graph* of a profile $\mathcal{P}$, denoted $H_{\mathcal{P}}$, is the graph obtained from the disjoint union of the underlying trees $T_1, \ldots, T_k$ by identifying nodes that have the same label. Multiple edges between the same pair of nodes are replaced by a single edge. See Figure 4.3.

$H_{\mathcal{P}}$ has $O(M_{\mathcal{P}})$ nodes and edges, and can be constructed in $O(M_{\mathcal{P}})$ time. By Assumption 1, there is a bijection between the labels in $L(\mathcal{P})$ and the nodes of $H_{\mathcal{P}}$. Thus, from this point forward, we refer to the nodes of $H_{\mathcal{P}}$ by their labels. It is easy to see that if $H_{\mathcal{P}}$ is not connected, then $\mathcal{P}$ decomposes into label-disjoint sub-profiles, and that $\mathcal{P}$ is compatible if and only if each sub-profile is compatible. Thus, without loss of generality, we shall assume the following.

Assumption 2. *$H_{\mathcal{P}}$ is connected.*

4.3.1 Positions

A *position* (for $\mathcal{P}$) is a vector $U = (U(1), U(2), \ldots, U(k))$, where $U(i) \subseteq L(\mathcal{T}_i)$, for each $i \in [k]$. Since labels may be shared among trees, we may have $U(i) \cap U(j) \neq \emptyset$, for $i, j \in [k]$ with $i \neq j$. For each $i \in [k]$, let $\mathrm{Desc}_i(U) = \{\ell : \ell' \leq_{\mathcal{T}_i} \ell, \text{ for some } \ell' \in U(i)\}$, and let $\mathrm{Desc}_{\mathcal{P}}(U) = \bigcup_{i \in [k]} \mathrm{Desc}_i(U)$.

A position U is *valid* if, for each $i \in [k]$,

(V1) if $|U(i)| \geq 2$, then the elements of $U(i)$ are siblings in $\mathcal{T}_i$ and

(V2) $\mathrm{Desc}_i(U) = \mathrm{Desc}_{\mathcal{P}}(U) \cap L(\mathcal{T}_i)$.

Lemma 16. *For any valid position U, $\mathcal{P}|\mathrm{Desc}_{\mathcal{P}}(U) = \{\mathcal{T}_1|\mathrm{Desc}_1(U), \mathcal{T}_2|\mathrm{Desc}_1(U), \ldots, \mathcal{T}_k|\mathrm{Desc}_k(U)\}$.*

Proof. By (V2), we have that $\mathcal{T}_i|\mathrm{Desc}_i(U)$ and $\mathcal{T}_i|\mathrm{Desc}_{\mathcal{P}}(U) \cap L(\mathcal{T}_i)$ are isomorphic, for each $i \in [k]$. The lemma then follows from the definition of $\mathcal{P}|\mathrm{Desc}_{\mathcal{P}}(U)$. □

For any valid position U, $H_{\mathcal{P}}(U)$ denotes the subgraph of $H_{\mathcal{P}}$ induced by $\mathrm{Desc}_{\mathcal{P}}(U)$.

Observation 1. *For any valid position U, $H_{\mathcal{P}}(U)$ is the subgraph of $H_{\mathcal{P}}$ obtained by deleting all labels in $V(H_{\mathcal{P}}) \setminus \mathrm{Desc}_{\mathcal{P}}(U)$, along with all incident edges.*

A valid position of special interest to us is U_{init}, the *root position*, defined as follows.

$$U_{\mathrm{init}} = (\phi_i^{-1}(r(T_1)), \phi_i^{-1}(r(T_2)), \ldots, \phi_i^{-1}(r(T_k))). \tag{4.1}$$

That is, for each $i \in [k]$, $U_{\mathrm{init}}(i)$ is a singleton containing only the label of $r(T_i)$. In Figure 4.3, $(U_{\mathrm{init}}(1), U_{\mathrm{init}}(2), U_{\mathrm{init}}(3)) = (\{1\}, \{2\}, \{g\})$. It is straightforward to verify that U_{init} is indeed valid, that $\mathrm{Desc}_{\mathcal{P}}(U_{\mathrm{init}}) = L(\mathcal{P})$, and that $H_{\mathcal{P}}(U_{\mathrm{init}}) = H_{\mathcal{P}}$.

4.3.2 Semi-Universal Labels

Let U be a valid position, and let ℓ be a label in U. Then, ℓ is *semi-universal in* U if $U(i) = \{\ell\}$, for every $i \in [k]$ such that $\ell \in L(\mathcal{T}_i)$. In Figure 4.3, labels 1 and 2 are semi-universal in U_{init}, but g is not, since g is in both $L(\mathcal{T}_2)$ and $L(\mathcal{T}_3)$, but $U_{\mathrm{init}}(2) \neq \{g\}$.

The term "semi-universal", borrowed from Pe'er et al. (18), derives from the following fact. Suppose that $\mathcal{P}$ is ancestrally compatible, that $\mathcal{T}$ is a tree that ancestrally displays $\mathcal{P}$, and that ℓ is a semi-universal label for some valid position U. Then, as we shall see, ℓ must label the root u_ℓ of a subtree of $\mathcal{T}$ that contains all the descendants of ℓ in $\mathcal{T}_i$, for every i such that $\ell \in L(\mathcal{T}_i)$. The qualifier "semi" is because this subtree may also contain labels that do not descend from ℓ in any input tree, but descend from some other semi-universal label ℓ' in U instead. In this case, ℓ' also labels u_ℓ. We exploit this property of semi-universal labels in our ancestral compatibility algorithm and its proof of correctness (see Section 4.4).

For each label $\ell \in L(\mathcal{P})$, let k_ℓ denote the number of input trees that contain label ℓ. We can obtain k_ℓ for every $\ell \in L(\mathcal{P})$ in $O(M_\mathcal{P})$ time during the construction of $H_\mathcal{P}$.

Lemma 17. *Let $U = (U(1), \ldots, U(k))$ be a valid position. Then, label ℓ is semi-universal in U if the cardinality of the set $J_\ell = \{i \in [k] : U(i) = \{\ell\}\}$ equals k_ℓ.*

Proof. By definition, $U(i) = \{\ell\}$, for every $i \in J_\ell$. Since $|J_\ell| = k_\ell$, the lemma follows. □

4.3.3 Successor Positions

For every $i \in [k]$ and every $\ell \in L(\mathcal{T}_i)$, let $\mathrm{Ch}_i(\ell)$ denote the set of children of ℓ in $L(\mathcal{T}_i)$. For a subset A of $L(\mathcal{T}_i)$, let $\mathrm{Ch}_i(A) = \bigcup_{\ell \in A} \mathrm{Ch}_i(\ell)$. Let U be a valid position, and S be the set of semi-universal labels in U. The *successor of U with respect to S* is the position U' defined as follows. For each $i \in [k]$,

$$U'(i) = \begin{cases} \mathrm{Ch}_i(\ell) & \text{if } U(i) = \{\ell\}, \text{ for some } \ell \in S, \\ U(i) & \text{otherwise.} \end{cases}$$

In Figure 4.3, the set of semi-universal labels in U_{init} is $S = \{1, 2\}$. Since $\mathrm{Ch}_1(1) = \{3, f\}$ and $\mathrm{Ch}_2(2) = \{e, f, g\}$, the successor of U_{init} is $U' = (\{3, f\}, \{e, f, g\}, \{g\})$.

Observation 2. *Let U be a valid position, and let U' be the successor of U with respect to the set S of semi-universal labels in U. Then, $H_\mathcal{P}(U')$ can be obtained from $H_\mathcal{P}(U)$ by doing the following for each $\ell \in S$: (1) for each $i \in [k]$ such that $U(i) = \{\ell\}$, delete all edges between ℓ and $\mathrm{Ch}_i(\ell)$; (2) delete ℓ.*

Let U be a valid position, and W be a subset of $\mathrm{Desc}_\mathcal{P}(U)$. Then, $U|W$ denotes the position $(U(1) \cap W, \ldots, U(k) \cap W)$. In Figure 4.3, the components of $H_\mathcal{P}(U')$, where U' is the successor of U_{init}, are $W_1 =$

$\{3, 4, a, b, c, d, e, g\}$ and $W_2 = \{f, h, i\}$. Thus, $U'|W_1 = (\{3\}, \{e, g\}, \{g\})$ and $U'|W_2 = (\{f\}, \{f\}, \emptyset)$. We have the following result.

Lemma 18. *Let U be a valid position, and S be the set of all semi-universal labels in U. Let U' be the successor of U with respect to S, and let $W_1, W_2, \ldots, W_p$ be the label sets of the connected components of $H_{\mathcal{P}}(U')$. Then, $U'|W_j$ is a valid position, for each $j \in [p]$.*

Proof. It suffices to argue that U' satisfies conditions (V1) and (V2). The lemma then follows from the fact that the connected components of $H_{\mathcal{P}}(U')$ are label-disjoint.

U' must satisfy condition (V1), since U does. Suppose $\ell \in S$. Then, for each $i \in [k]$ such that $\ell \in L(\mathcal{T}_i)$, $\mathrm{Desc}_i(U') = \mathrm{Desc}_i(U) \setminus \{\ell\}$ and $\mathrm{Desc}_{\mathcal{P}}(U') \cap L(\mathcal{T}_i) = (\mathrm{Desc}_{\mathcal{P}}(U) \cap L(\mathcal{T}_i)) \setminus \{\ell\}$. Thus, since (V2) holds for U, it also holds for U'. □

4.4 Testing Ancestral Compatibility

4.4.1 Overview of the Algorithm

`BuildNT` (Algorithm 4) is our algorithm for testing compatibility of semi-labeled trees. Its argument, U, is a valid position in $\mathcal{P}$ such that $H_{\mathcal{P}}(U)$ is connected. Line 1 computes the set S of semi-universal labels in U. If S is empty, then, as argued in Theorem 4 below, $\mathcal{P}|\mathrm{Desc}_{\mathcal{P}}(U)$ is incompatible, and, thus, so is $\mathcal{P}$. This fact is reported in Line 3. Line 4 creates a tentative root r_U, labeled by S, for the tree $\mathcal{T}_U$ for $L(U)$. Line 5 checks if S contains exactly one label ℓ, with no proper descendants. If so, by the connectivity assumption, ℓ must be the sole member of $\mathrm{Desc}_{\mathcal{P}}(U)$; that is, $L(U) = \ell$. Therefore, Line 6 simply returns the tree with a single node, labeled by $S = \{\ell\}$. Line 7 updates U, replacing it by its successor with respect to S. Let $W_1, W_2, \ldots, W_p$ be the connected components of $H_{\mathcal{P}}(U)$ after updating U. By Lemma 18, $U|W_j$ is a valid position, for each $j \in [p]$. Lines 8–12 recursively invoke `BuildNT` on $U|W_j$ for each $j \in [p]$, to determine if there is a tree t_j that ancestrally displays $\mathcal{P}|\mathrm{Desc}_{\mathcal{P}}(U \cap W_j)$. If any subproblem is incompatible, Line 12 reports that $\mathcal{P}$ is incompatible. Otherwise, Line 13 returns the tree obtained by making the t_js the subtrees of root r_U.

Next, we argue the correctness of `BuildNT`.

Algorithm 4: `BuildNT`(U)

Input: A valid position U for $\mathcal{P}$ such that $H_{\mathcal{P}}(U)$ is connected.
Output: A semi-labeled tree that ancestrally displays $\mathcal{P}' = \mathcal{P}|\mathrm{Desc}_{\mathcal{P}}(U)$, if $\mathcal{P}'$ is ancestrally compatible; `incompatible` otherwise.

1 Let $S = \{\ell \in U : \ell \text{ is semi-universal in } U\}$
2 **if** $S = \emptyset$ **then**
3 **return** `incompatible`
4 Create a node r_U with label set S
5 **if** $|S| = 1$ *and the single element of* S *has no proper descendants* **then**
6 **return** r_U
7 Replace U by the successor of U with respect to S
8 Let $W_1, W_2, \ldots, W_p$ be the connected components of $H_{\mathcal{P}}(U)$
9 **foreach** $j \in [p]$ **do**
10 Let $t_j =$ `BuildNT`$(U|W_j)$
11 **if** t_j *is not a tree* **then**
12 **return** `incompatible`
13 **return** the tree with root r_U and subtrees $t_1, \ldots, t_p$

4.4.2 Correctness

Lemma 19. *Let U be a valid position in $\mathcal{P}$. If* `BuildNT`(U) *returns a tree $\mathcal{T}_U$, then $\mathcal{T}_U$ is a phylogenetic tree such that $L(\mathcal{T}_U) = L(U)$.*

Proof. We use induction on $|L(U)|$. The base case, where $|L(U)| = 1$, is handled by Lines 5–6. In this case, $S = L(U) = \{\ell\}$ and `BuildNT`(U) correctly returns the tree consisting of a single node, labeled by $\{\ell\}$. Otherwise, let $W_1, \ldots, W_p$ be the connected components of $H_{\mathcal{P}}(U)$ in step 8. Since `BuildNT`(U) returns tree $\mathcal{T}_U$, it must be the case that, for each $j \in [p]$, the result t_j returned by the recursive call to `BuildNT`$(U|W_j)$ in step 10 is a tree. Since $|S| \geq 1$, we have $|L(W_j)| < |L(U)|$, for each $j \in [p]$. Thus, we can assume inductively that t_j is a phylogenetic tree for $L(W_j)$. Since $S \cup \bigcup_{j\in[p]} L(W_j) = L(U)$, the tree returned in step 13 is a phylogeny with species set $L(U)$. □

Theorem 4. *Let $\mathcal{P} = \{\mathcal{T}_1, \ldots, \mathcal{T}_k\}$ be a profile and let U_{init} be the root position, as defined in Equation (4.1). Then,* `BuildNT`(U_{init}) *returns either (i) a semi-labeled tree $\mathcal{T}$ that ancestrally displays $\mathcal{P}$, if $\mathcal{P}$ is ancestrally compatible, or (ii)* `incompatible` *otherwise.*

Proof. `BuildNT`(U_{init}) either returns a tree or `incompatible`. We consider each case separately.

(i) Suppose that $\texttt{BuildNT}(U_{\text{init}})$ returns a semi-labeled tree $\mathcal{T}$. By Lemma 19, $L(\mathcal{T}) = L(\mathcal{P})$. We now prove that $\mathcal{T}$ ancestrally displays $\mathcal{P}$. By Lemma 13, it suffices to show that $D(\mathcal{T}_i) \subseteq D(\mathcal{T})$ and $N(\mathcal{T}_i) \subseteq N(\mathcal{T})$, for each $i \in [k]$.

Consider any $(\ell, \ell') \in D(\mathcal{T}_i)$. Then, ℓ has a child ℓ'' in $\mathcal{T}_i$ such that $\ell'' \leq_{\mathcal{T}_i} \ell'$ — note that we may have $\ell'' = \ell$. There must be a recursive call to $\texttt{BuildNT}(U)$, for some valid position U, where ℓ is the set S of semi-universal labels obtained in Line 1. By Observation 2, label ℓ'', and thus ℓ', both lie in one of the connected components of the graph obtained by deleting all labels in S, including ℓ, and their incident edges from $H_{\mathcal{P}}(U)$. It now follows from the construction of $\mathcal{T}$ that $(\ell, \ell') \in D(\mathcal{T})$. Thus, $D(\mathcal{T}_i) \subseteq D(\mathcal{T})$.

Now, consider any $\{\ell, \ell'\} \in N(\mathcal{T}_i)$. Let v be the lowest common ancestor of $\phi_i(\ell)$ and $\phi_i(\ell')$ in $\mathcal{T}_i$ and let ℓ_v be the label of v. Then, ℓ_v has a pair of children, ℓ_1 and ℓ_2 say, in $\mathcal{T}_i$ such that $\ell_1 \leq_{\mathcal{T}_i} \ell$, and $\ell_2 \leq_{\mathcal{T}_i} \ell'$. Because $\texttt{BuildNT}(U_{\text{init}})$ returns a tree, there are recursive calls $\texttt{BuildNT}(U_1)$ and $\texttt{BuildNT}(U_2)$ for valid positions U_1 and U_2 such that ℓ_1 is semi-universal for U_1 and ℓ_2 is semi-universal for U_2. We must have $U_1 \neq U_2$; otherwise, $|U_1(i)| = |U_2(i)| \geq 2$, and, thus, neither ℓ_1 nor ℓ_2 is semi-universal, a contradiction. Further, it follows from the construction of $\mathcal{T}$ that we must have $\text{Desc}_{\mathcal{P}}(U_1) \cap \text{Desc}_{\mathcal{P}}(U_2) = \emptyset$. Hence, $\ell \parallel_{\mathcal{T}} \ell'$, and, therefore, $\{\ell, \ell'\} \in N(\mathcal{T})$.

(ii) Asssume, by way of contradiction, that $\texttt{BuildNT}(U_{\text{init}})$ returns `incompatible`, but that $\mathcal{P}$ is ancestrally compatible. By assumption, there exists a semi-labeled tree $\mathcal{T}$ that ancestrally displays $\mathcal{P}$. Since $\texttt{BuildNT}(U_{\text{init}})$ returns `incompatible`, there is a recursive call to $\texttt{BuildNT}(U)$ for some valid position U such that U has no semi-universal label, and the set S of Line 1 is empty.

By Lemma 14, $\mathcal{T}|\text{Desc}_{\mathcal{P}}(U)$ ancestrally displays $\mathcal{P}|\text{Desc}_{\mathcal{P}}(U)$. Thus, by Lemma 16, $\mathcal{T}|\text{Desc}_{\mathcal{P}}(U)$ ancestrally displays $\mathcal{T}_i|\text{Desc}_i(U)$, for every $i \in [k]$. Let ℓ be any label in the label set of the root of $\mathcal{T}|\text{Desc}_{\mathcal{P}}(U)$. Then, for each $i \in [k]$ such that $\ell \in L(\mathcal{T}_i)$, ℓ must be the label of the root of $\mathcal{T}_i|\text{Desc}_i(U)$. Thus, for each such i, $U(i) = \{\ell\}$. Hence, ℓ is semi-universal in U, a contradiction. □

4.4.3 An Iterative Versionfigures

We now present $\text{BUILDNT}_{\text{N}}$ (Algorithm 5), an iterative version of `BuildNT`, which lends itself naturally to an efficient implementation. $\text{BUILDNT}_{\text{N}}$ performs a breadth-first traversal of `BuildNT`'s recursion tree,

using a first-in first-out queue Q that stores pairs of the form (U, pred), where U is a valid position in $\mathcal{P}$ and pred is a reference to the parent of the node corresponding to U in the supertree built so far. Recursive calls are simulated by enqueuing pairs corresponding to subproblems. Next, we describe the main steps of our iterative algorithm.

BUILDNT$_{\mathrm{N}}$ initializes its queue to contain the starting position, U_{init}, with a null parent. It then proceeds to the **while** loop of Lines 3–14. Each iteration of the loop starts by dequeuing a valid position U, along with a reference pred to the potential parent for the subtree for $L(U)$ in the supertree. The body of the loop closely follows the steps performed by a call to `BuildNT`(U). Line 5 computes the set S of semi-universal labels in U. If S is empty, the algorithm reports that $\mathcal{P}$ is incompatible and terminates (Lines 6–7). The algorithm then creates a tentative root r_U labeled by S for the tree $\mathcal{T}_U$ for $L(U)$, and links r_U to its parent (Line 8). If S consists of exactly one element that has no proper descendants, we skip the rest of the current iteration of the **while** loop, and **continue** to the next iteration (Lines 9–10). Line 11 replaces U by its successor with respect to S. Lines 13–14 enqueue each of $U|W_1, U|W_2, \ldots, U|W_p$, along with r_U, for processing in a subsequent iteration. If the **while** loop terminates without any incompatibility being detected, the algorithm returns the tree with root $r_{U_{\mathrm{init}}}$.

Although the order in which BUILDNT$_{\mathrm{N}}$ processes connected components differs from that of `BuildNT` — breadth-first instead of depth-first — , it is straightforward to see that the effect is equivalent, and the proof of correctness of `BuildNT` (Theorem 4) applies to BUILDNT$_{\mathrm{N}}$ as well. We thus state the following without proof.

Theorem 5. *Let $\mathcal{P} = \{\mathcal{T}_1, \ldots, \mathcal{T}_k\}$ be a profile. Then,* BUILDNT$_{\mathrm{N}}(\mathcal{P})$ *returns either (i) a semi-labeled tree $\mathcal{T}$ that ancestrally displays $\mathcal{P}$, if $\mathcal{P}$ is ancestrally compatible, or (ii)* `incompatible` *otherwise.*

Let Q be BUILDNT$_{\mathrm{N}}$'s first-in first-out queue. In the rest of the paper, we will say that a valid position U *is in* Q if $(U, \mathrm{pred}) \in Q$, for some pred. Let H_Q be the subgraph of $H_{\mathcal{P}}$ induced by $\bigcup\{\mathrm{Desc}(U) : U \text{ is in } Q\}$. By Observation 1, H_Q is obtained from $H_{\mathcal{P}}(U_{\mathrm{bef}})$ through edge and node deletions.

Lemma 20. *At the start of any iteration of* BUILDNT$_{\mathrm{N}}$*'s* ***while*** *loop, the set of connected components of H_Q is $\{V(H_{\mathcal{P}}(U)) : U \text{ is in } Q\}$.*

Algorithm 5: BUILDNT$_{\mathrm{N}}(\mathcal{P})$

Input: A profile $\mathcal{P}$.

Output: A tree T that displays $\mathcal{P}$, if $\mathcal{P}$ is compatible; `incompatible` otherwise.

1 Construct $H_{\mathcal{P}}(U_{\text{init}})$
2 ENQUEUE$(Q, (U_{\text{init}}, \texttt{null}))$
3 **while** Q *is not empty* **do**
4 $\quad (U, \text{pred}) =$ DEQUEUE(Q)
5 $\quad$ Let $S = \{\ell \in U : \ell$ is semi-universal in $U\}$
6 $\quad$ **if** $S = \emptyset$ **then**
7 $\quad\quad$ **return** `incompatible`
8 $\quad$ Create a node r_U with label set S and set $\text{parent}(r_U) = \text{pred}$
9 $\quad$ **if** $|S| = 1$ *and the single element of* S *has no proper descendants* **then**
10 $\quad\quad$ **continue**
11 $\quad$ Replace U by the successor of U with respect to S
12 $\quad$ Let $W_1, W_2, \ldots, W_p$ be the connected components of $H_{\mathcal{P}}(U)$
13 $\quad$ **foreach** $j \in [p]$ **do**
14 $\quad\quad$ ENQUEUE$(Q, (U|W_j, r_U))$
15 **return** the tree with root $r_{U_{\text{init}}}$

Proof. The property holds at the outset, since, by Assumption 2, $H_{\mathcal{P}} = H_{\mathcal{P}}(U_{\text{init}})$ is a connected graph, and the only element of Q is $(U_{\text{init}}, \texttt{null})$. Assume that the property holds at the beginning of iteration l. Let (U, pred) be the element dequeued from Q in Line 4. Then, $H_{\mathcal{P}}(U)$ is connected. In place of (U, pred), Lines 13–14 enqueue $(U|W_j, r_U)$, for each $j \in [p]$, where, by construction, $H_{\mathcal{P}}(U|W_j)$ is a connected component of $H_{\mathcal{P}}(U)$. Thus, the property holds at the beginning of iteration $l + 1$. □

In other words, Lemma 20 states that each iteration of BUILDNT$_{\mathrm{N}}(\mathcal{P})$ deals with a subgraph of $H_{\mathcal{P}}$, whose connected components are in one-to-one correspondence with the valid positions stored in Q. This is illustrated by the example described next.

4.4.4 An Example

Figures 4.4–4.8 illustrate the execution of BUILDNT$_{\mathrm{N}}$ on the profile $\mathcal{P} = (\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3)$ of Figure 4.1. The figures show how the graph H_Q — initially equal to $H_{\mathcal{P}} = H_{\mathcal{P}}(U_{\text{init}})$ (Figure 4.3) — evolves as its edges and nodes are deleted.

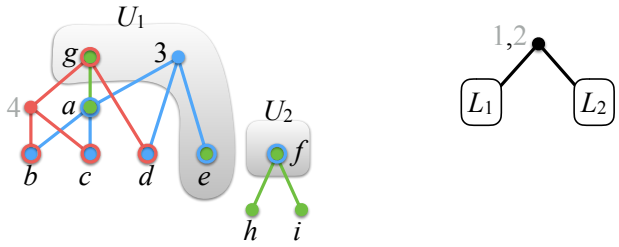

Figure 4.4 After generating all supertree nodes in level 0.

In each figure, H_Q is shown on the left and the current supertree is show on the right. For brevity, the figures only exhibit the state of H_Q and the supertree after all the nodes at each level are generated. The various valid positions that $\text{BUILDNT}_\text{N}(\mathcal{P})$ processes are denoted by U_α, for different subscripts α; S_α denotes the semi-universal labels in U_α, and U'_α denotes the successor of U_α with respect to S_α. We write L_α as an abbreviation for $L(U_\alpha)$ The root of the tree for L_α is r_{U_α} and is labeled by S_α.

Initially, $Q = ((U_{\text{init}},)$.

Level 0. Refer to Figure 4.4. As seen earlier, the set of semi-universal labels of U_{init} is $S_{\text{root}} = \{1, 2\}$. Thus, $H_\mathcal{P}(U'_{\text{init}})$ has two components W_1 and W_2. Let $U_1 = U'_{\text{init}}|W_1$ and $U_2 = U'_{\text{init}}|W_2$. Then,

$$U_1 = (\{3\}, \{e, g\}, \{g\}) \quad \text{and} \quad U_2 = (\{f\}, \{f\}, \emptyset).$$

After level 0 is processed, $Q = ((U_1, r_{U_{\text{init}}}), (U_2, r_{U_{\text{init}}}))$. This ensures that the roots of the subtrees for L_1 and L_2 will be children of $r_{U_{\text{init}}}$.

Level 1. Refer to Figure 4.5. We have $S_1 = \{3\}$, so $H_\mathcal{P}(U'_1)$ has two components W_{11} and W_{12}. Let $U_{11} = U'_1|W_{11}$ and $U_{12} = U'_1|W_{12}$. Then,

$$U_{11} = (\{a, d\}, \{g\}, \{g\}) \quad \text{and} \quad U_{12} = (\{e\}, \{e\}, \emptyset).$$

We have $S_2 = \{f\}$, so $H_\mathcal{P}(U'_2)$ has two components W_{21} and W_{22}. Let $U_{21} = U'_2|W_{21}$ and $U_{22} = U'_2|W_{22}$. Then,

$$U_{21} = (\emptyset, \{h\}, \emptyset) \quad \text{and} \quad U_{21} = (\emptyset, \{i\}, \emptyset).$$

After level 1 is processed, $Q = ((U_{11}, r_1), (U_{12}, r_1), (U_{21}, r_2), (U_{22}, r_2))$.

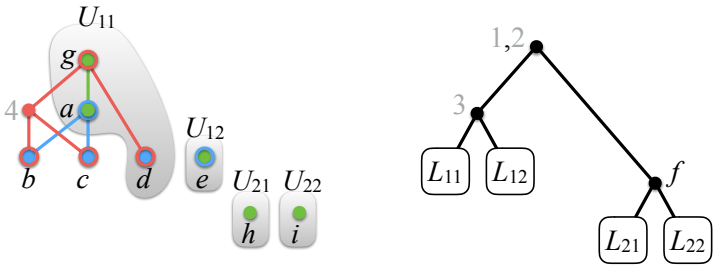

Figure 4.5 After generating all supertree nodes in level 1.

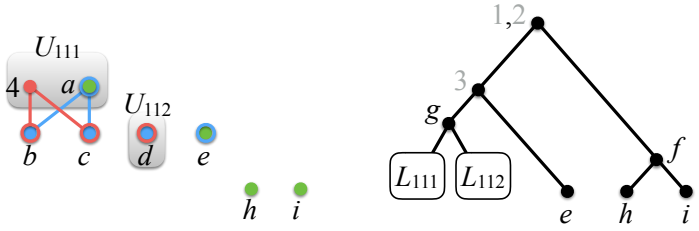

Figure 4.6 After generating all supertree nodes in level 2.

Level 2. Refer to Figure 4.6. We have $S_{11} = \{g\}$, so $H_{\mathcal{P}}(U'_{11})$ has two components W_{111} and W_{112}. Let $U_{111} = U'_{11}|W_{111}$ and $U_{112} = U'_{11}|W_{112}$. Then,

$$U_{111} = (\{a\}, \{a\}, \{4\}) \quad \text{and} \quad U_{112} = (\emptyset, \{d\}, \{d\}).$$

The only semi-universal labels in U_{12}, U_{21}, and U_{22} are, respectively, e, h, and i. Since none of these labels have proper descendants, each of them is a leaf in the supertree.

After level 2 is processed, $Q = ((U_{111}, r_{11}), (U_{112}, r_{11}))$.

Level 3. Refer to Figure 4.7. We have $S_{111} = \{4, a\}$, so $H_{\mathcal{P}}(U'_{111})$ has two components W_{1111} and W_{1112}. Let $U_{1111} = U'_{111}|W_{1111}$ and $U_{1112} = U'_{111}|W_{1112}$. Then,

$$U_{1111} = (\{b\}, \emptyset, \{b\}) \quad \text{and} \quad U_{1112} = (\{c\}, \emptyset, \{c\}).$$

The only semi-universal label in U_{112} is d. Since d has no proper descendants, it becomes a leaf in the supertree.

After level 3 is processed, $Q = ((U_{1111}, r_{111}), (U_{1112}, r_{111}))$.

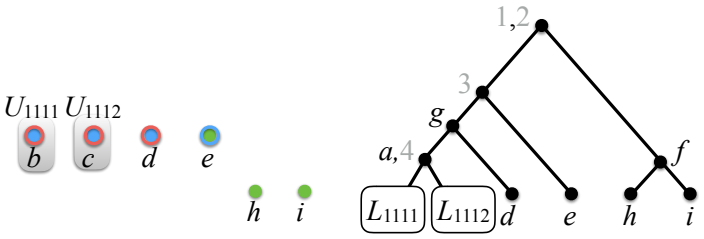

Figure 4.7 After generating all supertree nodes in level 3.

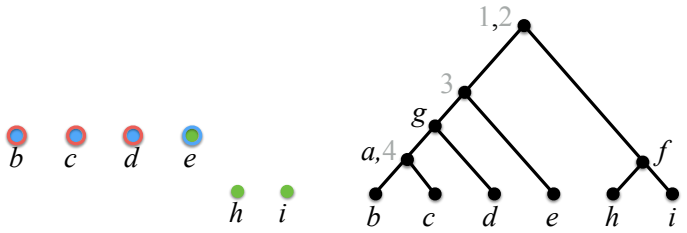

Figure 4.8 After generating all supertree nodes in level 4.

Level 4. Refer to Figure 4.8. The only semi-universal labels in U_{1111} and U_{1112} are, respectively, b and c. Since neither of these labels have proper descendants, each of them is a leaf in the supertree.

After level 4 is processed, Q is empty, and $\textsc{BuildNT}_{\text{N}}(\mathcal{P})$ terminates.

4.5 Implementation

Here we prove the following result.

Theorem 6. *There is an algorithm that, given a profile $\mathcal{P}$ of rooted trees, runs in $O(M_{\mathcal{P}} \log^2 M_{\mathcal{P}})$ time, and either returns a tree that displays $\mathcal{P}$, if $\mathcal{P}$ is compatible, or reports that is $\mathcal{P}$ is incompatible otherwise.*

We prove this theorem by showing how to implement $\textsc{BuildNT}_{\text{N}}$ so that the algorithm runs in $O(M_{\mathcal{P}} \log^2 M_{\mathcal{P}})$ on any profile $\mathcal{P}$.

As in Section 4.4.3, let H_Q denote the subgraph of $H_{\mathcal{P}}$ associated with the valid positions in $\textsc{BuildNT}_{\text{N}}$'s queue. By Lemma 20, each valid position U in Q corresponds to one connected component of H_Q, namely $\text{Desc}(U)$, and vice-versa. We use this fact in the implementation of $\textsc{BuildNT}_{\text{N}}$: alongside each valid position U in Q, we also store a reference to the respective connected component, along with additional information to quickly identify semi-universal labels.

Let U be any valid set in Q, let $Y = V(H_{\mathcal{P}}(U))$ be the corresponding connected component of H_Q, and let ℓ be any label in Y. Our implementation maintains the following data fields.

- Let $J_U = \{i \in [k] : U(i) \neq \emptyset\}$. Then, $Y.\texttt{map}$ is a map from J_U to $L(U)$, where, for each $i \in J_U$, $Y.\texttt{map}(i) = U(i)$.
- For each $\ell \in Y$, $\ell.\texttt{count}$ equals the cardinality of the set $\{i \in [k] : Y.\texttt{map}(i) = \{\ell\}\}$. (Recall that k_ℓ is the number of input trees that contain ℓ.)
- $Y.\texttt{semiU}$, a set consisting of all $i \in [k]$ such that $Y.\texttt{map}(i) = \{\ell\}$ for some $\ell \in Y$ such that $\ell.\texttt{count} = k_\ell$.
- $Y.\texttt{weight}$, which equals $\sum_{\ell \in Y} k_\ell$. This field is needed for technical reasons, to be explained later.

Now suppose that U is the valid position extracted from Q at the beginning of an iteration of the **while** loop of Lines 3–14, and that $Y = V(H_{\mathcal{P}}(U))$. Then, by Lemma 17, the set of semi-universal labels in U is $\{v \in Y.\texttt{map}(i) : i \in Y.\texttt{semiU}\}$. Thus, the data fields listed above allow us to find and retrieve the set S of line 5 of BUILDNT$_{\text{N}}(\mathcal{P})$ in $O(1)$ time. The task that remains is to devise an efficient way to update these fields for each of the connected components of $H_{\mathcal{P}}(U)$ created by replacing U with its successor in Line 11.

Let U_{bef} and U_{aft} be the values of U immediately before and after Line 11. Thus, U_{aft} is the successor of U_{bef}, and, by Observation 2, $H_{\mathcal{P}}(U_{\text{aft}})$ is obtained from $H_{\mathcal{P}}(U_{\text{bef}})$ through edge and node deletions. We need to

(a) generate the new connected components resulting from these deletions, and

(b) produce the required `map`, `count`, and `semiU` data fields for the various connected components.

We accomplish (a) using the dynamic graph connectivity data structure of Holm et al. (28), which we refer to as *HDT*. HDT allows us to maintain the list of nodes in each component, as well as the number of these nodes so that, if we start with no edges in a graph with N nodes, the amortized cost of each update is $O(\log^2 N)$. Since $H_{\mathcal{P}}$ has $O(M_{\mathcal{P}})$ nodes, each update takes $O(\log^2 M_{\mathcal{P}})$ time. The total number of edge and node deletions performed by BUILDNT$_{\text{N}}(\mathcal{P})$ — including all deletions in the interations — is at most the total number of edges and nodes in $H_{\mathcal{P}}$, which is $O(M_{\mathcal{P}})$. HDT allows us to maintain

connectivity information throughout the entire algorithm in $O(M_{\mathcal{P}} \log^2 M_{\mathcal{P}})$ time, which is within the time bound claimed in Theorem 6.

For part (b), we need to augment HDT in order to maintain the the required data fields for the various connected components created during edge and node deletion. In the next subsections, we describe how to do this. We begin by explaining how to initialize all the required data fields for $H_{\mathcal{P}} = H_{\mathcal{P}}(U_{\text{init}})$.

4.5.1 Initializing the Data Fields

Graph $H_{\mathcal{P}}(U_{\text{init}})$ has a single connected component, $Y_{\text{init}} = L(\mathcal{P})$, which is the entire vertex set of the graph. We initialize the data fields as follows.

- For each $i \in [k]$, $Y_{\text{init}}.\texttt{map}(i) = \{r(T_i)\}$. This takes $O(k)$ time.
- $Y_{\text{init}}.\texttt{weight} = \sum_{\ell \in L(\mathcal{P})} k_\ell$. This takes $O(M_{\mathcal{P}})$ time.

We initialize the `count` fields in $O(M_{\mathcal{P}})$ time as follows:

1. Set $\ell.\texttt{count}$ to 0 for all $\ell \in L(\mathcal{P})$.
2. For each $i \in [k]$, do the following.
   (a) Let ρ_i denote the label of $r(T_i)$.
   (b) For each each $j \in [k]$ such that $\rho_i \in L(T_j)$, increment $\rho_i.\texttt{count}$ by one if $Y_{\text{init}}.\texttt{map}(j) = \{\rho_i\}$.

Once the `count` fields are initialized, it is easy to initialize $Y_{\text{init}}.\texttt{semiU}$ in $O(k)$ time. Thus, we can initialize all the required fields in $O(M_{\mathcal{P}})$ time.

4.5.2 Maintaining the Data Fields

Suppose that all data fields fields are correctly computed for every connected component that is in Q at the beginning of an iteration of the **while** loop in 3–14 of BUILDNT$_{\text{N}}$. We now show how to generate the same fields efficiently for the new connected components created by Line 11.

4.5.2.1 Computing Successor Positions

Let U be the valid position extracted from Q at the beginning of an iteration of BUILDNT$_{\text{N}}$'s **while** loop, and let $Y = V(\text{Desc}(U))$ be the associated connected component. Assume all the data fields for Y have been correctly computed. To obtain the successor of U in Line 11 of BUILDNT$_{\text{N}}$, we perform the following steps.

1. Identify the set of semi-universal labels. As explained earlier, this set is given by $S = \{\ell \in Y.\texttt{map}(i) : i \in Y.\texttt{semiU}\}$.
2. Set $Y.\texttt{map}(i) = \emptyset$, for every $i \in Y.\texttt{semiU}$.
3. Make $Y.\texttt{semiU} = \emptyset$.
4. For each $\ell \in S$ and each i such that $\ell \in L(T_i)$, do the following.
   - If $\text{Ch}_i(\ell) \neq \emptyset$, replace $Y.\texttt{map}(i)$ by $\text{Ch}_i(\ell)$. If $\text{Ch}_i(\ell)$ is a singleton set $\{\alpha\}$, increment $\alpha.\texttt{count}$ by one. If $\alpha.\texttt{count} = k_\ell$, add i to $Y.\texttt{semiU}$.
   - Otherwise, $Y.\texttt{map}(i)$ is undefined.
5. For each label ℓ in S, delete the edges incident on ℓ and then ℓ itself, updating the data fields as necessary after each deletion.

The total number of updates to `map` and `semiU` fields in Steps 1–2 is $O(\sum_{\ell \in S} k_\ell)$. Since each label becomes semi-universal at most once, the total number of operations on `map` fields over the entire execution of BUILDNT$_{\text{N}}(\mathcal{P})$ is $O(\sum_{\ell \in L(\mathcal{P})} k_\ell)$, which is $O(M_{\mathcal{P}})$. The same bound holds for updates to `count` and `semiU` fields.

Next let us focus on how to handle the deletion of a single edge in Step 5.

4.5.2.2 Deleting an Edge

To delete an edge between ℓ and a child α of ℓ, we proceed as follows.

1. Delete (ℓ, α), querying HDT to determine whether this disconnects Y.

- If Y remains connected, skip the next steps and proceed directly to the next child of ℓ.
- Otherwise, Y is split into two components, Y_1 and Y_2.

2. Update Y_1.`weight` and Y_2.`weight`.
3. Identify which of Y_1 and Y_2 has the smaller `weight` field. Without loss of generality, assume that Y_1.`weight` $\leq$ Y_2.`weight`.
4. Initialize Y_1.`map` and Y_1.`semiU` to `null`.
5. Initialize Y_2.`map` and Y_2.`semiU` to Y.`map` and Y.`semiU`, respectively.
6. For each label β in Y_1, perform the following steps for each i such that $\beta \in L(\mathcal{T}_i)$.
   (a) Delete β from Y_2.`map`(i) and add β to Y_1.`map`(i).
   (b) Adjust `count` and `semiU` fields as necessary.

The connectivity test in Step 1 is done by querying HDT. Steps 3–5 are trivial. We thus focus on Steps 2 and 6.

To perform Step 2, we use the well-known technique of scanning the smaller component (10). We first consult HDT to determine which of Y_1 or Y_2 has fewer nodes. Assume, without loss of generality, that $|Y_1| \leq |Y_2|$. We initialize Y_1.`weight` to 0 and Y_2.`weight` to Y.`weight`. We then scan the labels of Y_1, incrementing Y_1.`weight` by k_ℓ for each label $\ell \in Y_1$. When the scan of Y_1 is complete, we make Y_2.`weight` $= Y_2$.`weight` $- Y_1$.`weight`. We claim that any label $\ell \in L(\mathcal{P})$ is scanned $O(\log M_\mathcal{P})$ times over the entire execution of BUILDNT$_\text{N}(\mathcal{P})$. To verify this, let $N(\ell)$ be the number of nodes in the connected component containing ℓ. Suppose that, initially, $N(\ell) = N$. Then, the rth time we scan ℓ, $N(\ell) \leq N/2^r$. Thus, ℓ is scanned $O(\log N)$ times. The claim follows, since $N = O(M_\mathcal{P})$. Therefore, the total number of updates over all labels is $O(M_\mathcal{P} \log M_\mathcal{P})$, and the work per update is $O(1)$.

Each execution of Step 6(a) updates each of Y_1.`map`(i) and Y_2.`map`(i) once. Step 6(b) is more complex, but can also be accomplished with $O(1)$ data field updates. We omit the (tedious) details.

Let us track the number of data field updates that can be attributed to some specific label $\beta \in L(P)$ over the entire execution of BUILDNT$_\text{N}(\mathcal{P})$. Each execution of step 6 for β performs $O(k_\beta)$ data field

updates. Let $w_r(\beta)$ be the weight of the connected component containing β at the beginning of step 6, at the rth time that β is considered in that step; thus, $w_0(\beta) \leq \sum_{\ell \in L(\mathcal{P})} k_\ell$. We claim that $w_r(\beta) \leq w_0(\beta)/2^r$. The reason is that we only consider β if (a) β is contained in one of the two components that result from deleting an edge in step 1 and (b) the component containing β has the smaller weight. Thus, the number of times β is considered in step 6 over the entire execution of BUILDNT$_{\text{N}}(\mathcal{P})$ is $O(\log w_0(\beta))$, which is $O(\log M_{\mathcal{P}})$, since $w_0(\beta) = O(M_{\mathcal{P}})$. Therefore, the total number of data field updates over all labels is $O(\log M_{\mathcal{P}} \cdot \sum_{\ell \in L(\mathcal{P})} k_\ell)$, which is $O(M_{\mathcal{P}} \log M_{\mathcal{P}})$.

4.5.3 Summary

Let us review the running times of each aspect of our implementation of BUILDNT$_{\text{N}}$.

- **Initializing the data structures.** This has two parts.
  - *Setting up the HDT data structure for $H_{\mathcal{P}}$.* This takes $O(M_{\mathcal{P}} \log^2 M_{\mathcal{P}})$ time.
  - *Initializing the data fields for the single connected component of $H_{\mathcal{P}}$.* This takes $O(M_{\mathcal{P}})$ time.
- **Maintaining the data structures.** This also has two parts.
  - *Updating the HDT data structure.* There are $O(M_{\mathcal{P}})$ edge and node deletions, at an amortized cost of $O(\log^2 M_{\mathcal{P}})$ per deletion, yielding a total time of $O(M_{\mathcal{P}} \log^2 M_{\mathcal{P}})$.
  - *Maintaining the relevant data fields for the connected components.* Assume, conservatively, that each update can be done in $O(\log M_{\mathcal{P}})$ time. Then, this step takes a total of $O(M_{\mathcal{P}} \log^2 M_{\mathcal{P}})$ over the entire execution of BUILDNT$_{\text{N}}$.

We conclude that the total running time of BUILDNT$_{\text{N}}(\mathcal{P})$ is $O(M_{\mathcal{P}} \log^2 M_{\mathcal{P}})$, completing the proof of Theorem 6.

4.6 Discussion

Like our earlier algorithm for compatibility of ordinary phylogenetic trees, the more general algorithm presented here, `BuildNT`, is a polylogarithmic factor away from optimality (a trivial lower bound is $\Omega(M_{\mathcal{P}})$,

the time to read the input). `BuildNT` has a linear-space implementation, using the results of Thorup (24). A question to be investigated next is the performance of the algorithm on real data. Another important issue is integrating our algorithm into a synthesis method that deals with incompatible profiles.

CHAPTER 5. CONCLUSIONS AND DISCUSSIONS

5.1 Conclusions

In this dissertation, we have presented a new graph-based approaches with $\tilde{O}(M_{\mathcal{P}})$ algorithms for the leaf-based and ancestral compatibility problems, where $M_{\mathcal{P}}$ is the total number of nodes and edges in the trees in $\mathcal{P}$. Unlike the best previous algorithm, the running time of our method does not depend on the degrees of the nodes in the input trees. We have also proved their correctness and analyzed their running time and implement details.

The main contributions of our work are:

For leaf-labeled trees compatibility checking, we present an $O(M_{\mathcal{P}} \log^2 M_{\mathcal{P}})$ algorithm that, given a collection $\mathcal{P}$ of phylogenetic trees, either returns a phylogeny that displays $\mathcal{P}$, if $\mathcal{P}$ is compatible, or reports that $\mathcal{P}$ is incompatible, otherwise. At first, we reviews basic concepts in phylogenetics and defines compatibility formally. And we introduces three useful graphs — the triple graph, the cluster intersection graph, and the display graph — and discusses their interrelationships. Then, we presents our intersection graph approach to testing tree compatibility. At last, we describes the implementation details needed to achieve the $O(M_{\mathcal{P}} \log^2 M_{\mathcal{P}})$ time complexity.

When the giving input trees are *semi-labeled trees*, that is, phylogenies whose internal nodes may be labeled by higher-order taxa, we give a $\tilde{O}(M_{\mathcal{P}})$ algorithm for ancestral compatibility problem, where $M_{\mathcal{P}}$ is the total number of nodes and edges in the trees in $\mathcal{P}$. Firstly, we presents basic definitions regarding semi-labeled trees and ancestral compatibility. Secondly, we introduces the display graph and discusses its properties. Thirdly, we presents `BuildNT`, our algorithm for testing ancestral compatibility. Fourthly, we gives the implementation details for `BuildNT`.

5.2 Discussions

Besides our first algorithm BUILDST$_{\mathrm{N}}$, for compatibility of ordinary phylogenetic trees, the more general algorithm also presented here, `BuildNT`, is a polylogarithmic factor away from optimality. `BuildNT` has a linear-space implementation, using the results of Thorup (24).

We know that a trivial lower bound for the tree compatibility problem is $\Omega(M_{\mathcal{P}})$, the time to read the input. Thus, our result leaves us a polylogarithmic factor away from an optimal algorithm for compatibility. Is it possible to reduce or even eliminate this gap? The bottleneck is the time to maintain the information associated with the various components of $H_{\mathcal{P}}(U)$. It is conceivable that the special structure of this graph and the way the deletions are performed could be used to our advantage. A second question is how well our algorithm performs in practice. To investigate this, it should be possible to leverage existing knowledge on the empirical behavior of dynamic connectivity data structures (16). Another important issue is integrating our algorithm into a synthesis method that deals with incompatible profiles.

BIBLIOGRAPHY

[1] A. V. Aho, Y. Sagiv, T. G. Szymanski, and J. D. Ullman. Inferring a tree from lowest common ancestors with an application to the optimization of relational expressions. *SIAM J. Comput.*, 10(3):405–421, 1981.

[2] B. R. Baum. Combining trees as a way of combining data sets for phylogenetic inference, and the desirability of combining gene trees. *Taxon*, 41:3–10, 1992.

[3] O. R. P. Bininda-Emonds, M. Cardillo, K. E. Jones, R. D. E. MacPhee, R. M. D. Beck, R. Grenyer, S. A. Price, R. A. Vos, J. L. Gittleman, and A. Purvis. The delayed rise of present-day mammals. *Nature*, 446:507–512, 2007.

[4] D. Bryant and J. Lagergren. Compatibility of unrooted phylogenetic trees is FPT. *Theoretical Computer Science*, 351:296–302, 2006.

[5] P. Buneman. A characterisation of rigid circuit graphs. *Discrete Math.*, 9:205–212, 1974.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[7] B. Courcelle. The monadic second-order logic of graphs I. Recognizable sets of finite graphs. *Inf. Comput.*, 85(1):12–75, 1990.

[8] Y. Deng and D. Fernández-Baca. Fast compatibility testing for phylogenies with nested taxa. In M. C. Frith and C. N. S. Pedersen, editors, *Algorithms in Bioinformatics - 16th International Workshop, WABI 2016, Aarhus, Denmark, August 22-24, 2016. Proceedings*, volume 9838 of *Lecture Notes in Computer Science*, pages 90–101. Springer, 2016.

[9] Y. Deng and D. Fernández-Baca. Fast compatibility testing for rooted phylogenetic trees. In *Proceedings of 27th Annual Symposium on Combinatorial Pattern Matching*, to appear.

[10] S. Even and Y. Shiloach. An on-line edge-deletion problem. *J. ACM*, 28(1):1–4, Jan. 1981.

[11] S. Grünewald, M. Steel, and M. S. Swenson. Closure operations in phylogenetics. *Mathematical Biosciences*, 208:521—537, 2007.

[12] M. R. Henzinger and V. King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *J. ACM*, 46(4):502–516, July 1999.

[13] M. R. Henzinger, V. King, and T. Warnow. Constructing a tree from homeomorphic subtrees, with applications to computational evolutionary biology. *Algorithmica*, 24:1–13, 1999.

[14] C. E. Hinchliff, S. A. Smith, J. F. Allman, J. G. Burleigh, R. Chaudhary, L. M. Coghill, K. A. Crandall, J. Deng, B. T. Drew, R. Gazis, K. Gude, D. S. Hibbett, L. A. Katz, H. D. Laughinghouse IV, E. J. McTavish, P. E. Midford, C. L. Owen, R. H. Reed, J. A. Reesk, D. E. Soltis, T. Williams, and K. A. Cranston. Synthesis of phylogeny and taxonomy into a comprehensive tree of life. *Proceedings of the National Academy of Sciences*, 112(41):12764–12769, 2015.

[28] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, July 2001.

[16] R. Iyer, D. Karger, H. Rahul, and M. Thorup. An experimental study of polylogarithmic, fully dynamic, connectivity algorithms. *J. Exp. Algorithmics*, 6, Dec. 2001.

[17] J. Jansson, C. Shen, and W. Sung. Improved algorithms for constructing consensus trees. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 1800–1813, 2013.

[18] I. Pe'er, T. Pupko, R. Shamir, and R. Sharan. Incomplete directed perfect phylogeny. *SIAM J. Comput.*, 33(3):590–607, 2004.

[19] M. A. Ragan. Phylogenetic inference based on matrix representation of trees. *Molecular Phylogenetics and Evolution*, 1:53–58, 1992.

[20] M. J. Sanderson. Phylogenetic signal in the eukaryotic tree of life. *Science*, 321(5885):121–123, 2008.

[21] C. Semple and M. Steel. *Phylogenetics*. Oxford Lecture Series in Mathematics. Oxford University Press, Oxford, 2003.

[22] Bininda-Emonds, Olaf R. P. *Phylogenetic Supertrees: Combining Information to Reveal the Tree of Life*. Ser. on Comp. Biol. Springer, 2003.

[23] M. A. Steel. The complexity of reconstructing trees from qualitative characters and subtrees. *J. Classification*, 9:91–116, 1992.

[24] M. Thorup. Near-optimal fully-dynamic graph connectivity. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing*, pages 343–350. ACM, 2000.

[25] V. Berry and C. Semple. Fast computation of supertrees for compatible phylogenies with nested taxa. *Systematic Biology*, 55(2):270–288, 2006.

[26] M. Bordewich, G. Evans, and C. Semple. Extending the limits of supertree methods. *Annals of Combinatorics*, 10:31–51, 2006.

[27] P. Daniel and C. Semple. Supertree algorithms for nested taxa. In O. R. P. Bininda-Emonds, editor, *Phylogenetic supertrees: Combining information to reveal the Tree of Life*, pages 151–171. Kluwer, Dordrecht, 2004.

[28] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, July 2001.

[29] R. M. Page. Taxonomy, supertrees, and the tree of life. In O. R. P. Bininda-Emonds, editor, *Phylogenetic supertrees: Combining information to reveal the Tree of Life*, pages 247–265. Kluwer, Dordrecht, 2004.

[30] E. W. Sayers et al. Database resources of the National Center for Biotechnology Information. *Nucleic Acids Research*, 37(Database issue):D5–D15, 2009.

[31] S. A. Smith, J. W. Brown, and C. E. Hinchliff. Analyzing and synthesizing phylogenies using tree alignment graphs. *PLoS Computational Biology*, 9(9):e1003223, 2013.

[32] The Angiosperm Phylogeny Group. An update of the Angiosperm Phylogeny Group classification for the orders and families of flowering plants: APG IV. *Botanical Journal of the Linnean Society*, 181:1–20, 2016.

[33] Stamatakis, Alexandros. RAxML version 8: A tool for phylogenetic analysis and post-analysis of large phylogenies. *Bioinformatics*, 30(9):1312–1313, 2014.

[34] Sanderson, Michael J. and Driskell, Amy C. and Ree, Richard H. and Eulenstein, Oliver and Langley, Sasha. Obtaining Maximal Concatenated Phylogenetic Data Sets from Large Sequence Databases . *Mol. Biol. Evol.*, 20(7):1036–1042, 2003.